

Cuarta edición
C/C++ CÓMO PROGRAMAR
y Java™
Incluye CD-ROM
con Microsoft Visual C++® 6.0
Introductory Edition
EN UN APURO ?
FALSO
VERDADERO
PEARSON
Prentice Hall
DEITEL®
DEITEL
DEITEL



CUARTA EDICIÓN

# CÓMO PROGRAMAR EN

# C/C++

# y Java



CUARTA EDICIÓN

# CÓMO PROGRAMAR EN




# y Java

Harvey M. Deitel
Deitel & Associates, Inc.

Paul J. Deitel
Deitel & Associates, Inc.

**TRADUCCIÓN**

**Jorge Octavio García Pérez**
*Ingeniero en Computación*
*Universidad Nacional Autónoma de México*

**REVISIÓN TÉCNICA**

**Arturo del Ángel Ramírez**
*Jefe de Departamento de la División de Sistemas*
*Facultad de Ingeniería Mecánica y Eléctrica*
*Universidad Autónoma de Nuevo León*

**M. en C. Gabriela Azucena Campos García**
*Profesora de tiempo completo*
*Departamento de Sistemas de Información, División de Profesional y Graduados*
*Instituto Tecnológico y de Estudios Superiores de Monterrey*
*Campus Estado de México*

**M. en C. Sergio Fuenlabrada Velázquez**
**Ing. Mario Alberto Sesma Martínez**
**Ing. Mario Oviedo Galdeano**
**Ing. Juan Alberto Segundo Miranda**
*Profesores Investigadores*
*Academia de Computación*
*Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas*
*Instituto Politécnico Nacional*




MÉXICO • ARGENTINA • BRASIL • COLOMBIA • COSTA RICA • CHILE • ECUADOR
ESPAÑA • GUATEMALA • PANAMÁ • PERÚ • PUERTO RICO • URUGUAY • VENEZUELA

Datos de catalogación bibliográfica

**DEITEL, HARVEY M. y DEITEL, PAUL J.**
**Cuarta edición**

**Cómo programar en C/C++ y Java**

PEARSON EDUCACIÓN, México, 2004

ISBN: 970-26-0531-8
Área: Universitarios

Formato: 20 × 25.5 cm Páginas: 1152



ISBN 0-13-142644-3



**Edición en español:**
Editor: Guillermo Trujano Mendoza
e-mail: guillermo.trujano@pearsoned.com
Editor de desarrollo: Miguel B. Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

**Edición en inglés:**
Vice President and Editorial Director: ***Marcia J. Horton***
Senior Acquisitions Editor: ***Kate Hargett***
Assistant Editor: ***Sarah Parker***
Editorial Assistant: ***Michael Giacobbe***
Vice President and Director of Production and Manufacturing, ESM: ***David W. Riccardi***
Executive Managing Editor: ***Vince O'Brien***
Managing Editor: ***Tom Manshreck***
Production Editor: ***Chirag Thakkar***
Production Editor, Media: ***Bob Engelhardt***
Director of Creative Services: ***Paul Belfanti***
Creative Director: ***Carole Anson***
Art Director: ***Geoff Cassar***
Chapter Opener and Cover Designer: ***Dr. Harvey Deitel and David Merrell***
Manufacturing Manager: ***Trudy Pisciotti***
Manufacturing Buyer: ***Lisa McDowell***
Marketing Manager: ***Pamela Shaffer***

CUARTA EDICIÓN, 2004



Cámara Nacional de la Industria Editorial Mexicana.
Reg. Núm. 1031.






ISBN 970-26-0531-8

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 07 06 05 04

*A Marcia Horton,*

*Directora Editorial de Ingeniería y Ciencias de la Computación en Prentice Hall:*

*Ha sido un privilegio y un placer elaborar el programa de publicaciones de Deitel contigo a lo largo de los últimos 18 años. Gracias por ser nuestra mentora y nuestra amiga.*

*Harvey y Paul Deitel*



# Contenido

# Prefacio

¡Bienvenido a ANSI/ISO C, C++ y Java Estándar! En Deitel & Associates escribimos tanto libros de texto de nivel universitario como libros profesionales sobre lenguajes de programación, y trabajamos arduamente para mantenerlos actualizados mediante un flujo constante de nuevas ediciones. Escribir la cuarta edición de este libro fue un placer. Este libro, así como su material de apoyo, contiene todo lo que los maestros y estudiantes necesitan para lograr una experiencia informativa, interesante, educativa, desafiante y entretenida. Pusimos a tono la escritura, la pedagogía, el estilo para codificar y el paquete de accesorios del libro. Además, en este prefacio incluimos un Recorrido a través del libro, el cual ayudará a los profesores, estudiantes y profesionales a tener una idea más clara de la amplia cobertura que este libro proporciona sobre la programación en C, C++ y Java.

En este prefacio planteamos las convenciones que utilizamos en este libro, tales como la presentación de la sintaxis de los códigos de ejemplo, el "lavado de código" y el resaltado de segmentos importantes de éste, para ayudar a que los estudiantes se enfoquen en los conceptos clave que se presentan en cada capítulo. También presentamos las nuevas características de la cuarta edición de *Cómo programar en C*.

El libro incluye el software de Microsoft, Visual C++® 6.0 Introductory Edition. Para brindar más apoyo a los programadores principiantes, ofrecemos varias de nuestras nuevas publicaciones de *Dive-Into™ Series*, las cuales pueden descargar gratuitamente desde **www.deitel.com**. Dicho material, en inglés, explica cómo compilar, ejecutar y depurar programas en C, C++ y Java, utilizando diversos entornos de desarrollo.

Aquí explicamos la suite completa de materiales educativos que apoyan a este libro, para ayudar a los profesores que utilicen este libro como texto en un curso a maximizar la experiencia educativa de sus estudiantes. Dicha suite incluye un CD, en inglés, llamado *Instructor's Resource*, el cual contiene las soluciones a los ejercicios de los capítulos del libro y un archivo llamado *Test-Item File* con cientos de preguntas de opción múltiple y sus respuestas. En el sitio Web de este libro (**www.peasoneducacion.net/deitel**), están disponibles recursos adicionales para el profesor, entre los cuales se incluyen el *Syllabus Manager* y *Lecture Notes*, diapositivas de PowerPoint. De igual manera los estudiantes, encontrará diapositivas de PowerPoint y material de apoyo adicional.

Este libro fue revisado por un equipo de académicos distinguidos y profesionales de la industria, que incluye a los principales miembros del comité de estándares de C; listamos sus nombres y sus lugares de trabajo para que tenga una idea de cuan cuidadosamente se examinó el libro. El prefacio concluye con información sobre los autores y sobre Deitel & Associates, Inc. Si al leer este libro le surge alguna duda, envíenos un correo electrónico a **deitel@deitel.com**; le responderemos de inmediato. Visite con regularidad nuestro sitio Web, **www.deitel.com**, e inscríbase en el boletín de noticias *Deitel® Buzz Online*, en **www.deitel.com/newsletter/subscribe.html**. Utilizamos el sitio Web y el boletín para mantener actualizados a nuestros lectores, con respecto a todas las publicaciones y servicios Deitel.

## *Características de* Cómo programar en C, cuarta edición

***Resaltado de código y de entradas de usuario***

Hemos agregado bastante código resaltado para facilitar a los lectores la identificación de los segmentos representativos de cada programa. Esta característica ayuda a los estudiantes a revisar rápidamente el material cuando se preparan para exámenes o para algún laboratorio. También resaltamos en nuestra pantalla los diálogos que los usuarios introducen, para diferenciarlos de las salidas de programa.

***"Lavado de código"***

"Lavado de código" es el término que utilizamos para aplicar comentarios, para utilizar identificadores importantes, para aplicar sangría y espaciado vertical que nos sirven para separar unidades importantes de un programa. Este proceso da como resultado programas que son más fáciles de leer y de autodocumentar. Hemos agregado comentarios amplios y descriptivos a todo el código, incluyendo un comentario antes y después de cada instrucción principal de control, para ayudar a que el estudiante comprenda claramente el flujo del programa. Le hicimos un buen "lavado" a todo el código fuente de los programas de este texto y de los accesorios.

Para promover buenas prácticas de programación, actualizamos todo el código fuente de los programas correspondientes a la parte de C de este libro con nuevos estándares de codificación. Las definiciones de variables ahora se ubican en líneas separadas para facilitar su lectura, y cada instrucción de control tiene una llave que abre y una que cierra, incluso si resulta redundante. Esto ayudará al lector cuando desarrolle programas largos y complejos. Cada prototipo de función ahora coincide con la primera línea de la definición de función, incluyendo los nombres de los parámetros (lo cual ayuda a documentar el programa y a reducir errores, en especial si se trata de programadores principiantes).

***Uso de terminología/presentación***

Hemos actualizado el uso de la terminología a lo largo del texto, para cumplir con los diversos estándares y especificaciones del lenguaje.

## *Método de enseñanza*

Muchos maestros creen que la complejidad de C, y muchas otras dificultades, hacen que este tema no es conveniente para un primer curso de programación; siendo que ese primer curso es precisamente el objetivo de este libro. Si no, ¿por qué habríamos escrito este libro?

Durante dos décadas, el Dr. Harvey M. Deitel impartió cursos introductorios a la programación a nivel universitario, en los que enfatizaba el desarrollo de programas claramente escritos y bien estructurados. Mucho de lo que se enseña en estos cursos son los principios básicos de la programación estructurada, con énfasis en el uso efectivo de instrucciones de control y en la funcionalidad. Nosotros presentamos este material exactamente en la misma forma en que Harvey Deitel lo hizo en sus cursos universitarios y los estudiantes se sienten motivados por el hecho de que aprenden un lenguaje que les será útil en cuanto entren en la industria.

Nuestro objetivo es claro: producir un libro de texto de programación en C para cursos introductorios de programación de nivel universitario, para estudiantes con poca o ninguna experiencia en el tema, pero que aun así ofrezca un riguroso y profundo tratamiento de la teoría y la práctica que exigen los cursos tradicionales de C. Para lograr estos objetivos hicimos un libro más grande que otros textos de C; esto se debe a que nuestro texto también enseña pacientemente los principios de la programación estructurada. Cientos de miles de estudiantes alrededor del mundo han aprendido C con ediciones anteriores de este libro.

Esta cuarta edición contiene una gran colección de ejemplos, ejercicios y proyectos sobre muchos campos, los cuales están diseñados para dar a los alumnos la oportunidad de resolver problemas reales muy interesantes, y el código de los ejemplos del texto fue probado en varios compiladores.

El libro se concentra en los principios de la buena ingeniería de software y hace hincapié en la claridad de los programas. Somos maestros que enseñamos temas de vanguardia en salones de clases de la industria alrededor del mundo y este texto pone énfasis en la buena pedagogía.

***Método del código activo (Método LIVE-CODE)***

Este libro contiene diversos ejemplos "reales"; cada nuevo concepto se presenta en el contexto de un programa completo, que funciona, y que es seguido de inmediato por una o más ejecuciones de ejemplo que muestran la

entrada/salida del programa. Este estilo ejemplifica la forma en que enseñamos y escribimos sobre programación. A este método de enseñanza y de escritura le llamamos método de código activo o Método ***LIVE-CODE***™. *Utilizamos lenguajes de programación para enseñar lenguajes de programación*. Leer los ejemplos que aparecen en este texto es muy parecido a escribirlos y ejecutarlos en una computadora.

***Acceso a World Wide Web***

Todo el código fuente (en inglés) de los ejemplos que aparecen en este libro (y en nuestras demás publicaciones) se encuentra disponible en Internet en el siguiente sitio Web:

`www.deitel.com`

Registrarse es rápido y sencillo, y las descargas son gratuitas. Modificar los ejemplos e inmediatamente ver los efectos de esos cambios es una excelente manera de mejorar su aprendizaje.

***Objetivos***

Cada capítulo comienza con una serie de objetivos que le informan al estudiante lo que debe esperar, y una vez que termina el capítulo, le brinda la oportunidad de determinar si los cumplió. Dicha serie de objetivos representa una base sólida y una fuente positiva de reafirmación.

***Frases***

Después de los objetivos de aprendizaje aparecen una o más frases; algunas son simpáticas, otras filosóficas, y las más ofrecen ideas interesantes. Hemos observado que los estudiantes disfrutan al relacionar las frases con el material del capítulo. Es probable que aprecie más algunas de las frases, *después* de leer los capítulos.

***Plan general***

El plan general del capítulo ayuda al estudiante a revisar el material de manera ordenada. Lo que también le ayuda a darse una idea de lo que verá, y a establecer un ritmo de aprendizaje cómodo y efectivo.

***Secciones***

Cada capítulo está organizado en pequeñas secciones que tratan temas clave de C, C++ o Java.

***13,280 líneas de en 268 programas de ejemplo (con los resultados del programa)***

Mediante nuestro método de código activo, presentamos características de C, C++ y Java en el contexto de programas completos que funcionan. Después de cada programa, aparece una ventana que contiene las salidas que se producen. Esto permite al estudiante confirmar que los programas funcionan como se esperaba. Relacionar las salidas de un programa con las instrucciones que producen dichas salidas es una excelente forma de aprender y de reforzar conceptos. Nuestros programas ejercitan muchas características de C, C++ y Java. Leer cuidadosamente el libro es parecido a introducir y ejecutar estos programas en una computadora.

***469 Ilustraciones/Figuras***

En este libro incluimos diversos diagramas, gráficos e ilustraciones. Las explicaciones que presentan los capítulos 3 y 4 sobre instrucciones de control muestran diagramas de flujo cuidadosamente dibujados. [*Nota:* Nosotros no enseñamos a utilizar diagramas de flujo como herramientas de desarrollo de programas, sin embargo, utilizamos una breve presentación orientada a los diagramas de flujo para especificar la precisa operación de las instrucciones de control de C.] El capítulo 12, Estructuras de datos, utiliza gráficos de líneas para ilustrar la creación y cómo mantener vinculadas listas, colas, pilas y árboles binarios. El resto del libro está bastante ilustrado.

***768 tips de programación***

Hemos incluido siete clases de tips de programación para ayudar a los estudiantes a que se enfoquen en aspectos importantes del desarrollo, prueba, depuración, rendimiento y portabilidad de los programas. Resaltamos cientos de estos tips como *Errores comunes de programación*, *Tips para prevenir errores*, *Buenas prácticas de programación*, *Observaciones de apariencia visual*, *Tips de rendimiento*, *Tips de portabilidad* y *Observaciones de ingeniería de software.* Estos tips y prácticas representan lo mejor de lo que hemos podido cosechar durante

seis décadas (combinadas) de experiencia docente y en programación. Una de nuestras alumnas, estudiante de matemáticas, nos dijo que pensaba que este método era como resaltar axiomas, teoremas y corolarios en libros de matemáticas, ya que proporciona una base sólida para construir un buen software.



### 259 Errores comunes de programación

*Los estudiantes que aprenden un lenguaje, en especial si se trata de su primer curso de programación, tienden a cometer con frecuencia ciertos errores. Poner atención en los apartados de* Errores comunes de programación *les ayuda a evitar cometer los mismos errores, y de paso reduce las largas filas afuera de la oficina del maestro.*



### 132 Buenas prácticas de programación

*Las* Buenas prácticas de programación *son tips para escribir programas claros. Estas técnicas ayudan a los estudiantes a producir programas más legibles, autodocumentados y fáciles de mantener.*



### 49 Tips para prevenir errores

*Cuando diseñamos por primera vez esta "clase de tip", pensamos que lo utilizaríamos estrictamente para decirle a la gente cómo probar y depurar programas, por lo que en ediciones anteriores se les conoció como "Tips de prueba y depuración". De hecho, muchos de los tips describen aspectos de C, C++ y Java que reducen la probabilidad de que se produzcan errores, lo que simplifica los procesos de prueba y depuración. Además, a lo largo del libro cambiamos muchas de las* Buenas prácticas de programación *por tips de esta clase.*




### 32 Observaciones de apariencia visual

*En la parte de Java de este libro proporcionamos* Observaciones de apariencia visual *para resaltar convenciones de interfaz gráfica de usuario. Estas observaciones ayudan a los estudiantes a diseñar sus propias interfaces gráficas de usuario para que cumplan con las normas de la industria.*



### 68 Tips de rendimiento

*Según nuestra experiencia, enseñar a los estudiantes a escribir programas claros y comprensibles es, por mucho, el objetivo más importante para un primer curso de programación. Sin embargo, los estudiantes quieren escribir programas que se ejecuten lo más rápidamente posible, que utilicen la menor cantidad de memoria, que necesiten el menor número de teclazos y que destaquen de alguna otra manera. Los estudiantes realmente se preocupan por el rendimiento y quieren saber qué pueden hacer para "mejorar" sus programas. Por lo tanto, resaltamos las oportunidades para mejorar el rendimiento de los programas, es decir, cuando hacemos que los programas se ejecuten más rápido o cuando minimizamos la cantidad de memoria que ocupan.*



### 38 Tips de portabilidad

*El desarrollo de software es una actividad compleja y cara. Las empresas que desarrollan software con frecuencia deben producir versiones personalizadas para una variedad de computadoras y sistemas operativos. Por ello, en la actualidad se pone gran énfasis en la portabilidad; es decir, en producir software que se ejecute en diversos sistemas operativos con pocos o ningún cambio. Mucha gente ofrece C, C++ y Java como lenguajes apropiados para el desarrollo de software portable. Algunas personas asumen que si implementan una aplicación en uno de los lenguajes, dicha aplicación automáticamente será portable. Éste simplemente no es el caso. Lograr la portabilidad requiere un diseño muy cuidadoso ya que existen muchas dificultades para ello. Nosotros incluimos varios* Tips de portabilidad *para ayudar a los estudiantes a escribir código portable. Desde su concepción, Java fue diseñado para maximizar la portabilidad, sin embargo, los programas en Java también pueden necesitar modificaciones para tener esa funcionalidad.*



### 189 Observaciones de ingeniería de software

*Las* Observaciones de ingeniería de software *resaltan las técnicas, las cuestiones arquitectónicas, los asuntos de diseño, etcétera, que afectan la arquitectura y construcción de los sistemas de software, en especial de los sistemas a gran escala. Mucho de lo que el estudiante aprenda aquí será útil en cursos más avanzados y en la industria, cuando comience a trabajar con sistemas reales grandes y complejos. C, C++ y Java son lenguajes de ingeniería de software especialmente efectivos.*

***Resumen***

Cada capítulo finaliza con elementos pedagógicos adicionales. En todos los capítulos presentamos un *Resumen* completo en forma de lista que ayuda al estudiante a revisar y a reforzar los conceptos clave. Cada capítulo contiene un promedio de 37 puntos de resumen.

***Terminología***

Incluimos una sección de terminología que contiene una lista en orden alfabético de los términos importantes definidos en el capítulo para reforzar aún más los conceptos. Cada capítulo contiene un promedio de 73 términos.

***Resumen de tips, prácticas y errores***

Al final de cada capítulo repetimos las *Buenas prácticas de programación*, los *Errores comunes de programación*, las *Observaciones de apariencia visual*, los *Tips de rendimiento*, los *Tips de portabilidad*, las *Observaciones de ingeniería de software* y los *Tips para prevención de errores*.

***728 Ejercicios de autoevaluación y sus respuestas (la cuenta incluye partes separadas)***

Incluimos amplias secciones de *Ejercicios de autoevaluación* y de *Respuestas a los ejercicios de autoevaluación* para que el alumno estudie por su cuenta. Esto le brindará la oportunidad de conocer el material y de prepararse para intentar los ejercicios regulares.

***993 Ejercicios (la cuenta incluye partes separadas; 1722 ejercicios en total)***

Cada capítulo finaliza con un conjunto importante de ejercicios que incluyen un sencillo repaso de la terminología y los conceptos importantes; la escritura de instrucciones específicas de un programa; la escritura de pequeñas partes o funciones y clases de C++/Java; la escritura de funciones completas, clases de C++/Java y programas; así como la escritura de proyectos finales importantes. El gran número de ejercicios permite a los profesores diseñar sus cursos de acuerdo con las necesidades específicas de sus alumnos, así como modificar las tareas del curso cada semestre. Los maestros pueden utilizar estos ejercicios para asignar tareas en casa, para aplicar exámenes cortos o para aplicar exámenes importantes.

***Un extenso índice***

Hemos incluido un extenso *Índice* al final del libro, el cual ayudará al estudiante a localizar cualquier término o concepto por palabra clave. El *Índice* es útil tanto para la gente que lee el libro por primera vez como para los programadores que ya ejercen y que utilizan el libro como referencia. La mayoría de los términos de las secciones de *Terminología* aparecen en el *Índice* (junto con muchas otras entradas de cada capítulo) por lo que el estudiante puede revisar estas secciones para asegurarse de que ha cubierto el material clave de cada capítulo.

## *Software incluido* con este libro

Al escribir este libro utilizamos varios compiladores de C. En su mayoría, los programas del texto funcionarán en todos los compiladores C de ANSI/ISO y de C++, incluyendo el compilador Visual C++ 6.0 Introductory Edition que acompaña a este libro.

El material de C (capítulos 2 a 14) sigue el ANSI C estándar publicado en 1990. Vea los manuales de referencia de su sistema para obtener más detalles sobre el lenguaje, o para obtener una copia del ANSI/ISO 9899: 1990, “American National Standard for Information Systems, Programming Language C”, del American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

En 1999, ISO aprobó una nueva versión de C, C99, la cual aún no es muy conocida. El Apéndice B contiene una lista completa de los recursos Web de C99. Si desea más información sobre C99 o le interesa adquirir una copia del documento de estándares de C99 (ISO/IEC 9899;1999), visite el sitio Web del American National Standards Institute (ANSI) en **`www.ansi.org`**.

El material de C++ está basado en el lenguaje de programación C++, tal como lo desarrolló el Comité acreditado de estándares INCITS, en su parte de tecnología de la información y su comité técnico J11, en el lenguaje de programación C++, respectivamente. La International Standards Organization (ISO) aprobó los lenguajes C y C++.

Todo buen programador debe leer cuidadosamente dichos documentos y revisarlos con frecuencia. Estos documentos no son manuales, sin embargo, definen sus respectivos lenguajes con el extraordinario nivel de precisión de quienes implementaron ese compilador y que los grandes desarrolladores requieren.

Los capítulos que manejan Java están basados en el lenguaje de programación Java de Sun Microsystems. Dicha empresa proporciona una implementación de Java 2 Platform llamada Java 2 Software Development Kit (J2SDK), el cual incluye el conjunto mínimo necesario de herramientas para escribir software en Java. Usted puede descargar la versión más reciente de J2SDK desde:

**java.sun.com/j2se/downloads.html**

La información relacionada con la instalación y configuración de J2SDK se encuentra en:

**developer.java.sun.com/developer/onlineTraining/new2java/gettingstartedjava.html**

Nosotros revisamos cuidadosamente nuestra presentación, comparándola con estos documentos. Nuestro libro pretende ser útil tanto en niveles introductorios como intermedios, por lo que no pretendimos cubrir todas las características analizadas en estos extensos documentos.

***Manuales de la serie DIVE-INTO™ para ambientes populares de C, C++ y Java***

Hemos lanzado nuestros nuevos manuales de la serie DIVE-INTO™ para ayudar a nuestros lectores a iniciarse en muchos de los ambientes de desarrollo de programas. Puede descargar gratuitamente estos manuales desde:

**www.deitel.com/books/downloads.html.**

Actualmente contamos con las siguientes publicaciones de la serie:

- *DIVE-INTO Microsoft® Visual C++ 6.*
- *DIVE-INTO Microsoft® Visual C++ .NET.*
- *DIVE-INTO Borland™ C++ Builder™ Compiler* (versión de línea de comandos).
- *DIVE-INTO Borland™ C++ Builder™ Personal* (versión IDE).
- *DIVE-INTO GNU C++ on Linux.*
- *DIVE-INTO GNU C++ via Cygwin on Windows* (Cygwin es un emulador UNIX para Windows que incluye el compilador GNU de C++).
- *DIVE-INTO Forte for Java Community Edition 3.0.*
- *DIVE-INTO SunOne Studio Community Edition 4.0.*

Cada uno de estos manuales muestra cómo compilar, ejecutar y depurar aplicaciones de C, C++ y Java en ese compilador en particular. Muchos de estos documentos también proporcionan instrucciones paso a paso, con instantáneas de la pantalla para ayudar a los lectores a instalar el software. Cada documento plantea información general sobre el compilador y la documentación en línea.

## *Paquete de accesorios para la cuarta edición de* Cómo programar en C

Este libro cuenta con diversas ayudas para los profesores. El CD llamado *Instructor's Resource* (*IRCD*) contiene el Manual del instructor con las soluciones a mayoría de los ejercicios que aparecen al final de cada capítulo. Este CD, en idioma inglés, está disponible únicamente para los profesores, a través de los representantes de Pearson Educación. **[*NOTA:* Por favor no nos escriba para solicitar este CD; su distribución está limitada estrictamente a profesores universitarios que utilicen este libro como texto en sus clases. Los profesores pueden obtener el manual de soluciones únicamente a través de los representantes de esta empresa.]** Los accesorios para este libro también incluyen un archivo llamado *Test Item File*, el cual contiene preguntas de opción múltiple. Además, pueden disponer de diapositivas de PowerPoint que contienen todo el código y las figuras del libro, así como una lista de los elementos que resumen los puntos clave del texto. Los profesores pueden adaptar estas diapositivas de acuerdo a sus necesidades. Pueden descargar estas diapositivas desde **www-.deitel.com** donde encontrarán recursos adicionales útiles tanto para profesores como para estudiantes. Adicionalmente, en el sitio Web Companion de este libro encontrará el Syllabus Manager, material que le ayudará a los profesores a planear sus cursos interactivamente y a crear programas de estudios en línea.

Los estudiantes también se ven beneficiados con la funcionalidad del sitio Web Companion. Los recursos específicos del libro para los estudiantes incluyen:

- Diapositivas de PowerPoint susceptibles de personalizar.
- Código fuente de todos los programas de ejemplo.

- Material de referencia de los apéndices del libro (tales como tablas de precedencia de operadores, conjuntos de caracteres y recursos Web).

Los recursos específicos de cada capítulo, disponibles para los estudiantes incluyen:

- Objetivos del capítulo.
- Lo más destacado (por ejemplo, el resumen del capítulo).
- Plan general.
- Tips (por ejemplo, *Errores comunes de programación*, *Buenas prácticas de programación*, *Tips de portabilidad*, *Tips de rendimiento*, *Observaciones de apariencia visual*, *Observaciones de ingeniería de software* y *Tips para prevenir errores*).
- La guía de estudio en línea, la cual contiene ejercicios adicionales de respuestas cortas para autoevaluación (por ejemplo, preguntas cuya respuesta es verdadero o falso) y sus respuestas, lo que proporciona al estudiante una retroalimentación inmediata.

El Sitio Web Companion con todo el material anterior, en idioma inglés, se encuentra en **`www.pearsoneducacion.net/deitel`**.

## *Iniciativas DEITEL® para aprendizaje electrónico*

***Libros electrónicos y soporte para dispositivos inalámbricos***

Los dispositivos inalámbricos tendrán un papel muy importante en el futuro de Internet. Dadas las recientes mejoras al ancho de banda y al surgimiento de tecnologías 2.5 y 3G, eso es lo que se vislumbra; dentro de unos cuantos años, más personas accederán a Internet por medio de dispositivos inalámbricos que por medio de computadoras de escritorio. Deitel & Associates está comprometida con la accesibilidad inalámbrica y hemos publicado *Wireless Internet & Mobile Business How to Program*. Estamos investigando nuevos formatos electrónicos, tales como libros electrónicos inalámbricos, para que los estudiantes y profesores tengan acceso al contenido virtualmente en cualquier momento y en cualquier lugar. Para enterarse de las actualizaciones periódicas de estas iniciativas, suscríbase al boletín de noticias *DEITEL® Buzz Online*, en **`www.deitel.com/newsletter/subscribe.html`**, o visite **`www.deitel.com`**.

## *Boletín de noticias DEITEL®* Buzz Online

Suscríbase a nuestro correo electrónico gratuito de noticias, *DEITEL Buzz Online*, que incluye comentarios sobre las tendencias y desarrollos de la industria, vínculos hacia artículos y recursos gratuitos de nuestras publicaciones actuales y futuras, calendarios de liberación de productos, erratas, retos, anécdotas, información sobre nuestros cursos empresariales de entrenamiento dirigidos por profesores, y mucho más. Para suscribirse visite:

**`www.deitel.com/newsletter/subscribe.html`**

## *La nueva serie para desarrolladores (DEITEL® Developer)*

Deitel & Associates, Inc., hizo el compromiso importante de cubrir las tecnologías de punta para los profesionales de la industria del software, a través del lanzamiento de nuestra *DEITEL® Developer Series*. Los primeros libros de la serie son *Web Services A Technical Introduction* y *Java Web Services for Experienced Programmers*. Estamos trabajando en *ASP .NET with Visual Basic .NET for Experienced Programmers*, *ASP .NET with C# for Experienced Programmers*, y en muchos más. Para saber sobre actualizaciones continuas de las publicaciones actuales y las venideras de la serie *DEITEL® Developer*, visite **`www.deitel.com`** o suscríbase a nuestro boletín de noticias.

## *Recorrido a través del libro*

El libro se divide en cuatro partes principales. La primera, capítulos 1 a 14, presenta un meticuloso tratamiento del lenguaje de programación C, el cual incluye una introducción formal a la programación estructurada. La segunda parte (capítulos 15 a 23), única entre los libros de texto de C, presenta un tratamiento completo sobre C++ y la programación orientada a objetos, suficiente para un curso universitario de posgrado. La terce-

ra parte (también única entre los libros de C), capítulos 24 a 30, presenta una introducción meticulosa a Java, la cual incluye programación de gráficos, programación de la interfaz gráfica de usuario (GUI) utilizando Java Swing, programación multimedia y programación basada en eventos. La cuarta parte, apéndices A a F, presenta una variedad de materiales de referencia que apoyan al texto principal.

***Parte 1: Programación por procedimientos en C***

**Capítulo 1 —Introducción a las computadoras, a Internet y a la World Wide Web—** Explica qué son las computadoras, cómo funcionan y cómo se programan. Introduce la idea de la programación estructurada y explica por qué este conjunto de técnicas motivaron una revolución en la forma de escribir los programas. El capítulo brinda una breve historia del desarrollo de los lenguajes de programación, desde los lenguajes máquina y los lenguajes ensambladores hasta los lenguajes de alto nivel; también explica los orígenes de C, C++ y Java. El capítulo incluye una introducción a los ambientes típicos de programación en C. Nosotros analizamos el gran interés que se ha suscitado en Internet con el advenimiento de la World Wide Web y el lenguaje de programación Java.

**Capítulo 2 —Introducción a la programación en C—** Proporciona una introducción concisa a la escritura de programas en C. Presenta un tratamiento detallado de las operaciones aritméticas y para la toma de decisiones en C. Después de estudiar este capítulo el estudiante sabrá cómo escribir programas sencillos, pero completos, en C.

**Capítulo 3 —Desarrollo de programas estructurados—** Tal vez éste sea el capítulo más importante del libro, en especial para estudiantes serios de ciencias de la computación. Éste introduce la idea de los algoritmos (procedimientos) para resolver problemas; explica la importancia de la programación estructurada para producir programas que sean claros, corregibles, que se puedan mantener y que probablemente funcionen al primer intento; introduce las instrucciones de control básicas de la programación estructurada, es decir, instrucciones de secuencia, de selección ( **`if`** e **`if…else`** ) y de repetición (**`while`**); explica la técnica de refinamiento arriba-abajo, paso a paso, que es importante para producir programas estructurados adecuados, y presenta la popular herramienta para programar, el pseudocódigo estructurado. Los métodos y técnicas utilizados en el capítulo 3 son aplicables a la programación estructurada en cualquier lenguaje de programación, no sólo en C. Este capítulo ayuda al estudiante a desarrollar buenos hábitos de programación y a prepararse para lidiar con tareas de programación más importantes a lo largo del libro.

**Capítulo 4 —Control de programas en C—** Mejora las nociones de la programación estructurada e introduce instrucciones adicionales de control. Examina detalladamente la repetición y compara los ciclos controlados por un contador y los ciclos controlados por centinelas. Introduce la instrucción **`for`** como un medio conveniente para implementar ciclos controlados por contador; presenta la instrucción de selección **`switch`** y la instrucción de repetición **`do…while`**. El capítulo concluye con una explicación de los operadores lógicos.

**Capítulo 5 —Funciones en C—** Explica el diseño y la construcción de módulos de programa. Las capacidades relacionadas con las funciones en C incluyen funciones de la biblioteca estándar, funciones definidas por el programador, recursividad y capacidades de llamadas por valor. Las técnicas que presentamos en el capítulo 5 son básicas para producir y apreciar los programas estructurados adecuadamente, en especial los programas grandes y el software que los programadores de sistemas y de aplicaciones podrían desarrollar en la realidad. Presentamos la estrategia de "divide y vencerás" como un medio efectivo para resolver problemas complejos, dividiéndolos en componentes más sencillos que interactúan entre sí. Los estudiantes disfrutan el tratamiento de números aleatorios y la simulación, y aprecian la explicación del juego de azar con dados, el cual utiliza de manera elegante las instrucciones de control. En este capítulo introducimos la enumeración, y en el capítulo 10 proporcionamos una explicación más detallada. El capítulo 5 ofrece una sólida introducción a la recursividad, e incluye una tabla que resume docenas de ejemplos de recursividad y ejercicios distribuidos en el resto del libro. Algunos libros dejan la recursividad para un capítulo posterior; sin embargo, nosotros pensamos que es mejor cubrir este tema de manera gradual a lo largo del texto. Los diversos ejercicios incluyen varios problemas clásicos de recursividad como el de la torre de Hanoi.

**Capítulo 6 —Arreglos en C—** Explica la estructuración de datos en arreglos, o grupos de elementos de datos relacionados del mismo tipo. El capítulo presenta diversos ejemplos, tanto de un solo subíndice, como de dos subíndices. Es bien sabido que estructurar datos de manera adecuada es tan importante como utilizar efectivamente instrucciones de control al desarrollar programas bien estructurados. Los ejemplos investigan

distintas formas comunes de manipulación de arreglos, la impresión de histogramas, el ordenamiento de datos, el paso de arreglos a funciones, y una introducción al campo del análisis de encuestas (con estadística simple). Una característica de este capítulo es la cuidadosa explicación de las técnicas elementales de ordenamiento y búsqueda, y la presentación de la búsqueda binaria como una enorme mejora de la búsqueda lineal. Los ejercicios que aparecen al final del capítulo incluyen diversos problemas interesantes y desafiantes, como las técnicas mejoradas de ordenamiento, el diseño de un sistema de reservaciones para una aerolínea, una introducción al concepto de los gráficos de tortuga (que se hicieron famosos gracias al lenguaje LOGO), y los problemas del recorrido del caballo y las ocho reinas, que muestran la idea de la programación heurística, la cual se utiliza ampliamente en el campo de la inteligencia artificial.

**Capítulo 7 —Apuntadores en C—** Presenta una de las características más poderosas y difíciles de dominar del lenguaje C: los apuntadores. El capítulo proporciona explicaciones detalladas acerca de los operadores para apuntadores, de las llamadas por referencia, de las expresiones con apuntadores, de la aritmética con apuntadores, de la relación entre apuntadores y arreglos, de los arreglos de apuntadores y de los apuntadores a funciones. Los ejercicios del capítulo incluyen una encantadora simulación de la clásica carrera entre la tortuga y la liebre, barajar y repartir cartas y cómo manejar algoritmos y recorridos recursivos a través de laberintos. También incluimos una sección especial llamada "Cómo construir su propia computadora". Esta sección explica la programación en lenguaje máquina y continúa con un proyecto que involucra el diseño y la implementación de un simulador de una computadora que permite al lector escribir y ejecutar programas en lenguaje máquina. Esta característica única del libro le será especialmente útil a aquel lector que desee comprender cómo funcionan en realidad las computadoras. Nuestros estudiantes disfrutan este proyecto y a menudo implementan mejoras sustanciales, muchas de las cuales se las sugerimos en los ejercicios. En el capítulo 12, otra sección especial guía al lector a través de la construcción de un compilador; el lenguaje máquina que produce el compilador se ejecuta después en el simulador de lenguaje máquina producido en el capítulo 7.

**Capítulo 8 —Caracteres y cadenas en C—** Trata de los fundamentos del procesamiento de datos no numéricos. El capítulo incluye un recorrido a través de las funciones para procesamiento de caracteres y cadenas, disponibles en las bibliotecas de C. Las técnicas que explicamos aquí se utilizan ampliamente en la construcción de procesadores de palabras, en software para diseño y composición de páginas, y en aplicaciones de procesamiento de texto. El capítulo incluye una variedad de ejercicios que exploran las aplicaciones de procesamiento de texto. El estudiante disfrutará los ejercicios sobre escritura de poemas humorísticos de cinco versos, escritura de poemas al azar, conversión del español a latín vulgar, generación de palabras de siete letras que equivaldrían a un número telefónico dado, justificación de texto, protección de cheques, escritura del monto de un cheque en palabras, generación de código Morse, conversiones métricas y letras de cambio. El último ejercicio reta al estudiante a utilizar un diccionario computarizado para crear un generador de crucigramas.

**Capítulo 9 — Formato de datos de entrada/salida en C—** Presenta todas las poderosas capacidades de formato de **`printf`** y **`scanf`**. Aquí explicamos las capacidades de **`printf`** para el formato de resultados, tales como redondeo de valores de punto flotante a un número dado de lugares decimales, alineación de columnas de números, justificación a la derecha y a la izquierda, inserción de información literal, cómo forzar un signo de suma, impresión de ceros, uso de notación exponencial, uso de números octales y hexadecimales, y control de anchos de campo y precisiones. Explicamos todas las secuencias de escape de **`printf`** para el movimiento del cursor, la impresión de caracteres especiales y cómo ocasionar una alerta audible. Examinamos todas las capacidades de **`scanf`** para el formato de datos de entrada, incluyendo la entrada de tipos específicos de datos y cómo evitar caracteres específicos en un flujo de entrada. Explicamos todos los especificadores de conversión de **`scanf`** para la lectura de valores decimales, octales, hexadecimales, de punto flotante, de carácter y de cadena. También explicamos la introducción de datos para que coincidan (o no) con los caracteres de un conjunto. Los ejercicios del capítulo virtualmente prueban todas las capacidades de formato para datos de entrada/salida.

**Capítulo 10 —Estructuras, uniones, manipulaciones de bits y enumeraciones en C—** Presenta diversas características importantes. Las estructuras son como los registros en otros lenguajes de programación, los cuales agrupan elementos de datos de varios tipos. En el capítulo 11 utilizamos las estructuras para formar archivos que consisten en registros de información. En el capítulo 12, utilizamos las estructuras junto con los apuntadores y la asignación dinámica de memoria para formar estructuras dinámicas de datos, como listas ligadas, colas, pilas y árboles. Las uniones permiten que un área de memoria sea utilizada por diferentes tipos

de datos en diferentes momentos; compartir la memoria de este modo puede reducir los requerimientos de memoria de un programa o sus requerimientos de almacenamiento secundario. Las enumeraciones proporcionan un medio conveniente para definir constantes simbólicas útiles; esto ayuda a escribir programas más autodocumentados. Las poderosas capacidades para la manipulación de bits en C permiten a los programadores escribir programas que ejerciten capacidades de hardware de más bajo nivel. Esto ayuda a los programas a procesar cadenas de bits, encender o apagar bits específicos y a almacenar información de manera más compacta. Dichas capacidades, que con frecuencia sólo se encuentran en lenguajes ensambladores de bajo nivel, son valoradas por programadores que escriben software de sistemas como sistemas operativos y software para redes. Una característica del capítulo es la simulación revisada y de alto rendimiento de cómo barajar y repartir cartas. Ésta es una excelente oportunidad para el profesor para enfatizar la calidad de los algoritmos.

**Capítulo 11 —Procesamiento de archivos en C—** Explica las técnicas utilizadas para el procesamiento de archivos de texto con acceso secuencial y acceso aleatorio. El capítulo comienza con una introducción a la jerarquía de datos como bits, bytes, campos, registros y archivos. Después presenta la visión de C con respecto a los archivos y los flujos. Explica los archivos de acceso secuencial utilizando programas que muestran cómo abrir y cerrar archivos, cómo almacenar datos en un archivo de manera secuencial, y cómo leer los datos de un archivo de manera secuencial. También explica los archivos de acceso aleatorio utilizando programas que muestran cómo crear un archivo de manera secuencial para acceso aleatorio, cómo leer y escribir datos en un archivo con acceso aleatorio, y cómo leer datos de manera secuencial desde un archivo al que se accedió de manera aleatoria. El cuarto programa de acceso aleatorio combina muchas de las técnicas de acceso a archivos, tanto secuencial como aleatorio, en un programa completo de procesamiento de transacciones.

**Capítulo 12 —Estructuras de datos en C—** Explica las técnicas utilizadas para crear y manipular estructuras de datos dinámicas. El capítulo comienza con explicaciones sobre las estructuras autorreferenciadas y la asignación dinámica de memoria, y continúa con una explicación sobre cómo crear y mantener distintas estructuras de datos dinámicas, las cuales incluyen listas ligadas, colas (o líneas de espera), pilas y árboles. Para cada tipo de estructura de datos presentamos programas completos y funcionales, y mostramos ejemplos de los resultados. El capítulo ayuda a los estudiantes a dominar los apuntadores. Incluye muchos ejemplos que utilizan la indirección (o desreferencia) y la doble indirección, un concepto particularmente difícil. Uno de los problemas al trabajar con apuntadores es que a los estudiantes se les dificulta visualizar las estructuras de datos y cómo se entrelazan sus nodos. El ejemplo del árbol binario es una maravillosa conclusión al estudio de los apuntadores y de las estructuras de datos dinámicas. Este ejemplo crea un árbol binario, refuerza la eliminación de duplicados, e introduce los recorridos recursivos del árbol en preorden, inorden y posorden. Los estudiantes tienen un sentido genuino de la responsabilidad cuando estudian e implementan este ejemplo; particularmente aprecian el poder ver que el recorrido inorden despliega los valores de los nodos en orden. El capítulo incluye una amplia colección de ejercicios. Lo más destacado de los ejercicios es la sección especial de “Cómo construir su propio compilador”. Los ejercicios guían al estudiante a través del desarrollo de un programa de conversión de expresiones de infijo a posfijo, y de un programa de evaluación de expresiones posfijo. Después modificamos el algoritmo de evaluación de expresiones posfijo para generar código en lenguaje máquina. El compilador coloca este código en un archivo (utilizando las técnicas del capítulo 11). Los estudiantes pueden ejecutar el lenguaje máquina producido por sus compiladores en los simuladores de software que construyeron en los ejercicios del capítulo 7.

**Capítulo 13 —El preprocesador de C—** Proporciona explicaciones detalladas sobre las directivas del preprocesador. El capítulo incluye información sobre la directiva **`#include`** (la cual ocasiona que se incluya una copia del archivo especificado en la posición de la directiva en el archivo de código fuente, antes de que el archivo se compile) y la directiva **`#define`** que crea constantes simbólicas y macros. El capítulo explica la compilación condicional para permitir al programador controlar la ejecución de las directivas del preprocesador y la compilación del código del programa. También explica el operador **`#`**, el cual convierte su operando en una cadena, y el operador **`##`** que concatena dos tokens. Aquí presentamos constantes simbólicas predefinidas, tales como **`_LINE_`**, **`_FILE_`**, **`_DATE_`** y **`_TIME_`**. Por último presentamos la macro **`assert`** del archivo de encabezado **`assert.h`**. La macro **`assert`** es muy valiosa en la evaluación, depuración, verificación y validación de programas.

**Capítulo 14 —Otros temas de C—** Presenta temas adicionales que incluyen diversos conceptos que por lo general no se cubren en cursos introductorios. Nosotros mostramos cómo redirigir la entrada de programas

para que provengan de un archivo, cómo redirigir la salida de un programa para que se ubique en un archivo, cómo redirigir la salida de un programa para que sea la entrada de otro (a lo que se le llama "canalización"), también a añadir la salida de un programa a un archivo existente, a desarrollar funciones que utilicen listas de argumentos de longitud variable, a pasar argumentos de líneas de comandos a la función **main** y utilizarlos en un programa, a compilar programas cuyos componentes se encuentran en múltiples archivos, a registrar funciones con **atexit** para que se ejecuten al terminar el programa, a terminar la ejecución de un programa con la función **exit**, cómo utilizar los calificadores de tipo **const** y **volatile**, cómo especificar el tipo de una constante numérica mediante los sufijos de entero y de punto flotante, a utilizar la biblioteca de manejo de señales para atrapar eventos inesperados, cómo crear y utilizar arreglos dinámicos con **calloc** y **realloc**, y a utilizar uniones como una técnica para ahorrar espacio.

***Parte 2: Programación basada y orientada a objetos y programación genérica en C++***

**Capítulo 15 —C++ como un "mejor C"—** Presenta las características no orientadas a objetos de C++. Estas características mejoran el proceso de escribir programas por procedimientos. El capítulo explica los comentarios de una sola línea, el flujo de entrada/salida, las declaraciones, la creación de nuevos tipos de datos, los prototipos de función y la verificación de tipo, las funciones **inline** (como reemplazo de macros), los parámetros por referencia, el calificador **const**, la asignación dinámica de memoria, los argumentos predeterminados, el operador unario de resolución de alcance, la sobrecarga de funciones, las especificaciones de enlazado y las plantillas de funciones.

**Capítulo 16 —Las clases de C++ y la abstracción de datos—** Comienza nuestra explicación sobre la programación orientada a objetos. El capítulo representa una maravillosa oportunidad para enseñar la abstracción de datos de "manera correcta", es decir, a través de un lenguaje (C++) expresamente dedicado a implementar tipos de datos abstractos (ADTs, Abstract Data Types). En años recientes, la abstracción de datos se ha vuelto un tema importante en los cursos de computación introductorios. Los capítulos 16 a 18 incluyen un tratamiento sólido de la abstracción de datos. El capítulo 16 explica la implementación de ADTs como las clases (**class**) de estilo de C++ y por qué este método supera al uso de **structs**; también explica cómo acceder a miembros **class**, cómo separar la interfaz de la implementación, cómo utilizar funciones de acceso y de utilería, cómo inicializar objetos con constructores, cómo destruir objetos con destructores, cómo asignar de manera predeterminada la copia de un miembro de un objeto, y la reutilización de software. Uno de los ejercicios del capítulo desafía al lector a desarrollar una clase para números complejos.

**Capítulo 17 —Las clases de C++. Parte II—** Continúa con el estudio de las clases y la abstracción de datos. El capítulo explica cómo declarar y utilizar objetos constantes, funciones miembro constantes, la composición (el proceso de construir clases que tienen como miembros a objetos de otras clases), funciones y clases **friend**, las cuales tienen derechos especiales de acceso a los miembros **private** y **protected** de las clases, el apuntador **this**, el cual permite a un objeto saber su propia dirección, la asignación dinámica de memoria, miembros **static** de la clase para contener y manipular todos los datos de la clase, ejemplos de populares tipos de datos abstractos (arreglos, cadenas y colas), clases contenedoras e iteradores. Los ejercicios del capítulo piden al estudiante que desarrolle una clase para cuentas de ahorros y una clase para almacenar conjuntos de números enteros. También explicamos la asignación dinámica de memoria con **new** y **delete**. En C++ estándar, cuando **new** falla, éste regresa un apuntador 0. Nosotros utilizamos este estilo estándar en los capítulos 17 a 22. Dejamos para el capítulo 23 la explicación del nuevo estilo de la falla de **new**, en la que ahora **new** "arroja una excepción". Motivamos la explicación de los miembros **static** de la clase con un ejemplo que se basa en un videojuego. A lo largo del libro y en nuestros seminarios profesionales enfatizamos la importancia de esconder los detalles de implementación a los clientes de una clase.

**Capítulo 18 —Sobrecarga de operadores en C++—** Éste es uno de los temas más populares de nuestros cursos de C++. Los estudiantes realmente disfrutan este material, ya que coincide perfectamente con la explicación de los tipos de datos abstractos de los capítulos 16 y 17. La sobrecarga de operadores permite a los programadores indicar al compilador cómo utilizar operadores existentes con objetos de nuevos tipos de clases. C++ ya sabe cómo utilizar estos operadores con objetos de tipos predefinidos, tales como enteros, números de punto flotante y caracteres. Sin embargo, suponga que creamos una nueva clase llamada cadena; ¿qué significaría el signo +, si se utilizara entre objetos de tipo cadena? Muchos programadores utilizan el signo + con cadenas para que indique una concatenación. El capítulo explica los fundamentos de la sobrecarga de opera-

dores, las restricciones de dicha sobrecarga, la sobrecarga con funciones miembro de la clase frente a la sobrecarga con funciones no miembro, la sobrecarga de operadores unarios y binarios, y la conversión de tipos. Una característica del capítulo es un importante ejemplo práctico que incluye una clase arreglo, una clase para enteros muy grandes y una clase para números complejos (las dos últimas aparecen con todo el código fuente en los ejercicios). Este material difiere de lo que generalmente se hace en los lenguajes de programación y de lo que se presenta en la mayoría de los cursos. La sobrecarga de operadores es un tema complejo, sin embargo, es muy enriquecedor. Utilizar inteligentemente la sobrecarga de operadores le ayuda a dar "estilo" a sus clases. Con las técnicas de los capítulos 16, 17 y 18, es posible crear una clase **`Fecha`** que, si la hubiéramos utilizado en las dos últimas décadas, habríamos podido eliminar fácilmente una parte importante del llamado "problema del año 2000". Uno de los ejercicios anima al lector a aumentar la sobrecarga de operadores a la clase **`Complejo`** para lograr una buena manipulación de los objetos de esta clase con símbolos de operadores (como en matemáticas), en lugar de utilizar llamadas a funciones, como el estudiante hizo en los ejercicios del capítulo 17.

**Capítulo 19 —Herencia en C++—** Trata con una de las capacidades fundamentales de los lenguajes de programación orientada a objetos. La herencia es una forma de reutilización de software, en la que las nuevas clases se desarrollan rápidamente y absorben fácilmente las capacidades de clases existentes y agregan de manera adecuada nuevas capacidades. El capítulo explica las nociones de las clases base y de las clases derivadas, los miembros **`protected`**, la herencia **`public`**, **`protected`** y **`private`**, las clases base directas, las clases base indirectas, los constructores y destructores en clases base y en clases derivadas, y la ingeniería de software con herencia. El capítulo compara la herencia (relación *es un*) con la composición (relación *tiene un*), e introduce las relaciones *utiliza un* y *conoce a*. Una característica del capítulo es que presenta muchos ejemplos prácticos importantes. En particular, un ejemplo que implementa la jerarquía de la clase punto, círculo, cilindro. El ejercicio pide al estudiante que compare la creación de nuevas clases por medio de herencia, con las creadas por medio de la composición, para que amplíe las diferentes jerarquías de herencia que explicamos en el capítulo, para que escriba una jerarquía de herencia para cuadriláteros, trapezoides, paralelogramos, rectángulos y cuadrados, y para que genere una jerarquía más general de formas bidimensionales y tridimensionales.

**Capítulo 20 —Funciones virtuales y polimorfismo en C++—** Trata con otra de las capacidades fundamentales de la programación orientada a objetos, es decir, con el comportamiento polimórfico. Cuando muchas clases están relacionadas con una clase base común a través de la herencia, cada objeto de clase derivada debe tratarse como un objeto de clase base. Esto permite que los programas se escriban de una manera general e independiente de los tipos específicos correspondiente a los objetos de clase derivada. Es posible manejar nuevos tipos de objetos con el mismo programa, lo que hace que los programas puedan ampliarse. El polimorfismo permite a los programas eliminar la compleja lógica de **`switches`** (indicadores), a favor de una lógica más sencilla en "línea recta". Por ejemplo, el administrador de pantalla de un videojuego puede enviar un mensaje de dibujo a cada objeto de una lista ligada de objetos a dibujarse. Cada objeto sabe cómo dibujarse a sí mismo. Es posible agregar un nuevo objeto al programa sin modificarlo, siempre y cuando ese nuevo objeto sepa cómo dibujarse a sí mismo. Este estilo de programación por lo general se utiliza para implementar las interfaces gráficas de usuario más populares de hoy en día. El capítulo explica la mecánica para lograr un comportamiento polimórfico a través de las funciones **`virtuales`**. Aquí se hace la distinción entre las clases abstractas (desde las cuales no se pueden obtener instancias para objetos) y las clases concretas (desde las que se pueden obtener instancias para objetos). Las clases abstractas son útiles para proporcionar una interfaz heredable a las clases, a través de toda la jerarquía. Una característica del capítulo es su ejemplo práctico sobre el polimorfismo de la jerarquía del punto, círculo y cilindro que explicamos en el capítulo 19. Los ejercicios del capítulo piden al estudiante que explique algunas cuestiones conceptuales y métodos, que añada clases abstractas a la jerarquía de formas y que desarrolle un paquete básico de gráficos mediante funciones **`virtuales`** y programación polimórfica. Nuestra audiencia profesional insistió en que explicáramos de manera precisa cómo se implementa el polimorfismo en C++, y qué "costos" de memoria y tiempo de ejecución uno debe pagar cuando se programa con esta poderosa capacidad. Nosotros respondimos desarrollando una ilustración en la sección titulada Polimorfismo, funciones **`virtuales`** y vinculación dinámica "Bajo la cubierta", que muestra las *vtables* (tablas de funciones virtuales) que el compilador de C++ construye automáticamente para apoyar el estilo de programación polimórfico. Nosotros dibujamos estas tablas en las clases en las que explicamos la jerarquía de formas punto, círculo y cilindro. Nuestras audiencias expresaron que esto les proporcionó la informa-

ción para decidir que el polimorfismo era un estilo de programación apropiado para cada nuevo proyecto que enfrentaran. Incluimos esta presentación en la sección 20.9 y la ilustración de la *vtable* en la figura 20.2. Estudie cuidadosamente esta presentación, ya que ésta le ayudará a comprender mejor lo que ocurre en la computadora cuando programe con herencia y polimorfismo.

**Capítulo 21 —Entrada/salida de flujo en C++—** Contiene un completo tratamiento de entradas/salidas orientadas a objetos de C++. El capítulo explica las diferentes capacidades en E/S de C++, incluyendo resultados con el operador de inserción de flujo, entradas con el operador de extracción de flujo, E/S con seguridad de tipo (una buena mejora sobre C), E/S con formato, E/S sin formato (para rendimiento), manipuladores de flujo para controlar la base numérica del flujo (decimal, octal o hexadecimal), números de punto flotante, control de anchos de campo, manipuladores definidos por el usuario, estados de formato de flujo, errores de estado de flujo, E/S de objetos de tipos definidos por el usuario y vinculación de flujos de salida con flujos de entrada (para garantizar que los indicadores de comandos realmente aparezcan antes de solicitar al usuario que introduzca una respuesta). El amplio conjunto de ejercicios pide al estudiante que escriba varios programas que prueben la mayoría de las capacidades de E/S que explicamos en el texto.

**Capítulo 22 —Plantillas de C++—** Explica una de las más recientes adiciones a C++. En el capítulo 15 presentamos las plantillas de funciones. Las plantillas de clases permiten al programador capturar la esencia de un tipo de dato abstracto (como pilas, arreglos o colas), y crear, con una mínima adición de código, versiones de ese ADT (tipo de dato abstracto) para tipos particulares (como una cola de **enteros**, una cola de **flotantes**, una cola de cadenas, etcétera). Por esta razón, las plantillas de clases con frecuencia se conocen como tipos parametrizados. El capítulo explica el uso de parámetros de tipo y sin tipo, y considera la interacción entre plantillas y otros conceptos de C++, como herencia y miembros **friend** y **static**. Los ejercicios desafían al estudiante a escribir una variedad de plantillas de funciones y de plantillas de clase, y a emplearlas en programas completos.

**Capítulo 23 —Manejo de excepciones en C++—** Explica una de las más recientes mejoras al lenguaje C++. El manejo de excepciones permite al programador escribir programas que son más fuertes, más tolerantes a fallas y más apropiados para ambientes de negocios críticos. El capítulo explica cuándo es adecuado el manejo de excepciones; presenta los fundamentos del manejo de excepciones mediante bloques **try**, instrucciones **throw** y bloques **catch**; indica cómo y cuándo relanzar una excepción; explica cómo escribir la especificación de una excepción y cómo procesar excepciones inesperadas; y explica los importantes vínculos entre las excepciones y los constructores, los destructores y la herencia. Explicamos el relanzamiento de excepciones e ilustramos las dos formas en que **new** puede fallar cuando la memoria está agotada. Antes del anteproyecto de C++ estándar, **new** fallaba y devolvía un 0, así como en C, cuando **malloc** falla devuelve un apuntador **NULL**. Mostramos el nuevo estilo de la falla de **new**, mediante el lanzamiento de una excepción **bad_alloc** (mala asignación). Mostramos cómo utilizar **set_new_handler** para especificar una función personalizada, a la que se llamará para lidiar con situaciones de agotamiento de memoria. Explicamos la plantilla de clase **auto_ptr** para garantizar que la memoria asignada de manera dinámica sea adecuadamente eliminada para evitar fugas de memoria.

***Parte 3: Programación orientada a objetos, interfaz gráfica de usuario manejada por eventos y programación multimedia y de gráficos en Java***

**Capítulo 24 —Introducción a aplicaciones y subprogramas de Java—** Presenta un ambiente de programación típico de Java y proporciona una ligera introducción a aplicaciones de programación y subprogramas (applets) en el lenguaje de programación Java. Algunas de las entradas y salidas se llevan a cabo mediante un nuevo elemento de interfaz gráfica de usuario (GUI) llamado **JOptionPane** que proporciona ventanas predefinidas (llamadas diálogos) para entrada y salida. **JOptionPane** maneja la salida de datos hacia ventanas y la entrada de datos desde ventanas. El capítulo presenta los subprogramas de Java utilizando muchos de los ejemplos que vienen con el Java 2 Software Development Kit (J2SDK). Nosotros utilizamos el visor de subprogramas (**appletviewer**) (una utilería que viene con el J2SDK) para ejecutar diversos ejemplos de subprogramas. Después escribimos subprogramas de Java que realizan tareas parecidas a las aplicaciones escritas al principio del capítulo, y explicamos las similitudes y las diferencias entre éstos y las aplicaciones. Después de estudiar este capítulo, el estudiante entenderá cómo escribir sencillas, pero completas, aplicaciones y subprogramas (applets) de Java. Los siguientes capítulos utilizan tanto subprogramas como aplicaciones para demostrar conceptos adicionales de programación.

**Capítulo 25 —Más allá de C y C++: operadores, métodos y arreglos—** Se enfoca tanto en las similitudes como en las diferencias que existen entre Java, C y C++. El capítulo explica los tipos primitivos en Java y en qué difieren de C/C++, así como algunas diferencias en terminología. Por ejemplo, lo que en C/C++ se conoce como función, en Java se conoce como método. El capítulo también contiene una explicación sobre los operadores lógicos: **&&** (AND lógico), **&** (AND lógico booleano), || (OR lógico), | (OR lógico booleano incluyente), ^(OR lógico booleano excluyente), y aplicaciones ! (NOT). Motivamos y explicamos el tema de la sobrecarga de métodos (como una comparación con la sobrecarga de funciones de C++). En este capítulo también presentamos eventos y manejo de eventos (elementos requeridos para programar interfaces gráficas de usuario). Los eventos son notificaciones de cambios de estado como el clic de botones, el clic del ratón, el oprimir alguna tecla, etcétera. Java permite a los programadores responder a diferentes eventos, codificando métodos llamados manejadores de eventos. También presentamos arreglos en Java, los cuales se procesan como objetos hechos y derechos. Esto representa una evidencia adicional del compromiso de Java de casi un 100% de orientación a objetos. Analizamos la estructuración de datos en arreglos, o grupos de elementos relacionados del mismo tipo. El capítulo presenta diversos ejemplos tanto de arreglos con un solo subíndice como de arreglos con dos subíndices.

**Capítulo 26 —Programación basada en objetos en Java—** Comienza nuestra explicación más a fondo sobre clases. El capítulo se enfoca en la esencia y en la terminología de las clases y los objetos. ¿Qué es un objeto?, ¿qué es una clase de objetos?, ¿cómo luce el interior de un objeto?, ¿cómo se crean los objetos?, ¿cómo se destruyen?, ¿cómo se comunican los objetos entre sí?, ¿por qué las clases son como un mecanismo natural para empacar software como componentes reutilizables? El capítulo explica la implementación de tipos de datos abstractos como clases de estilo Java, el acceso a miembros de la clase, cómo forzar el ocultamiento de información con variables de instancias **private**, cómo separar la interfaz de la implementación, cómo utilizar métodos de acceso y de utilidad, la inicialización de objetos mediante constructores, y el uso de constructores sobrecargados. El capítulo también explica la declaración y el uso de referencias constantes, la composición (el proceso de construir clases que tienen como miembros referencias hacia objetos), la referencia **this** que permite a un objeto "conocerse a sí mismo", la asignación dinámica de memoria, los miembros **static** de una clase para que contengan y manipulen datos de la clase, y ejemplos de tipos de datos abstractos populares, como pilas y colas. El capítulo también presenta la instrucción **package**, y explica cómo crear paquetes reutilizables. Los ejercicios del capítulo retan al estudiante a desarrollar clases para números complejos, números racionales, horas, fechas, rectángulos, enteros grandes, una clase para jugar gato, una clase para cuentas de ahorros y una clase para mantener conjuntos de enteros.

**Capítulo 27 —Programación orientada a objetos en Java—** Explica las relaciones entre clases de objetos, y la programación con clases relacionadas. ¿Cómo podemos aprovechar las similitudes entre clases de objetos para minimizar el trabajo necesario para construir sistemas de software grandes? ¿Qué es el polimorfismo? ¿Qué significa "programar en general", en lugar de "programar en específico"? ¿Cómo es que programar en general facilita la modificación de sistemas y la adición de nuevas características con un mínimo esfuerzo? ¿Cómo podemos programar para toda una categoría de objetos, en lugar de programar individualmente para cada tipo de objeto? El capítulo lidia con una de las capacidades más importantes de los lenguajes de programación orientada a objetos, la herencia, que es una forma de reutilización de software en la que las nuevas clases se desarrollan rápida y fácilmente, absorbiendo las capacidades de clases existentes y agregando nuevas capacidades adecuadas. El capítulo explica las nociones de las superclases y las subclases, de miembros **protected**, de superclases directas, de superclases indirectas, del uso de constructores en superclases y subclases, y de la ingeniería de software con herencia. Nosotros presentamos clases internas que ayudan a esconder detalles de implementación. Las clases internas se utilizan con mayor frecuencia para crear manejadores de eventos de la GUI. Las llamadas clases internas pueden declararse dentro de otras clases, y son útiles para definir manejadores de eventos comunes para diversos componentes de la GUI. Las clases internas anónimas se declaran dentro de métodos, y se utilizan para crear un objeto, por lo general un manejador de eventos para un componente específico de la GUI. El capítulo compara la herencia (relaciones *es un*) con la composición (relaciones *tiene un*). Una característica del capítulo es el ejemplo práctico que presenta sobre la implementación de una jerarquía de clases punto, círculo, cilindro. El ejercicio pide al estudiante que compare la creación de nuevas clases mediante herencia y mediante composición, que amplíe las jerarquías de herencia que explicamos en el capítulo, que escriba una jerarquía de herencia para cuadriláteros, trapezoides, paralelogramos,

rectángulos y cuadrados, y que genere una jerarquía más general de formas bidimensionales y tridimensionales. El capítulo explica el comportamiento polimórfico. Cuando muchas clases están relacionadas a través de la herencia con una superclase común, cada objeto de la subclase puede tratarse como un objeto de la superclase. Esto permite que los programas se escriban de manera general e independiente de los tipos específicos de los objetos de la subclase. Es posible manejar nuevos tipos de objetos con el mismo programa, lo que hace que los programas puedan ampliarse. El polimorfismo permite a los programas eliminar la compleja lógica de **`switches`** (indicadores), a favor de una lógica más sencilla en "línea recta". Por ejemplo, el administrador de pantalla de un videojuego puede enviar un mensaje de dibujo a cada objeto de una lista ligada de objetos a dibujarse. Cada objeto sabe cómo dibujarse a sí mismo. Es posible agregar un nuevo objeto al programa sin modificarlo, siempre y cuando ese nuevo objeto sepa cómo dibujarse a sí mismo. Este estilo de programación por lo general se utiliza para implementar las interfaces gráficas de usuario más populares de hoy en día. El capítulo hace la distinción entre las clases **`abstractas`** (desde las cuales no se pueden obtener instancias para objetos) y las clases concretas (desde las que se pueden obtener instancias para objetos). El capítulo también introduce las interfaces (conjuntos de métodos que deben ser definidos por cualquier clase que **`implemente`** la interfaz).

**Capítulo 28 —Gráficos en Java y Java2D—** Comienza una secuencia de tres capítulos que presenta la "chispa" multimedia de Java. La programación tradicional en C y C++ está bastante limitada al modo de caracteres de entrada/salida. Algunas versiones de C++ se apoyan en bibliotecas de clases que dependen de la plataforma, las cuales pueden hacer gráficos; sin embargo, si utiliza estas bibliotecas, es posible que sus aplicaciones no sean portables. Las capacidades de los gráficos de Java son independientes de la plataforma y, por lo tanto, son portables, y decimos portables en toda la extensión de la palabra. Usted puede desarrollar subprogramas de Java con muchos gráficos, y distribuirlos a sus colegas por la World Wide Web a cualquier parte, y ellos podrán ejecutarlos bien en las plataformas locales de Java. Nosotros explicamos contextos gráficos y objetos gráficos; dibujar cadenas, caracteres y bytes; control de colores y fuentes; manipulación de pantalla y modos de pintura; y trazado de líneas, rectángulos, redondeado de rectángulos, rectángulos tridimensionales, óvalos, arcos y polígonos. Presentamos la API Java2D, nueva en Java 2, el cual proporciona poderosas herramientas de manipulación de gráficos. El capítulo tiene muchas figuras que minuciosamente ilustran cada una de estas capacidades gráficas con ejemplos de código activo, con atractivos resultados en pantalla, con tablas características detalladas y con dibujos lineales detallados.

**Capítulo 29 —Componentes de la interfaz gráfica de usuario de Java—** Presenta la creación de subprogramas (applets) y aplicaciones con interfaces gráficas de usuario (GUIs) amigables con el usuario. Este capítulo se enfoca en los nuevos componentes Swing de la GUI de Java. Estos componentes de la GUI, independientes de la plataforma, están completamente escritos en Java. Esto proporciona componentes Swing con una gran flexibilidad; pueden personalizarse para que se parezcan a la plataforma de la computadora en la que el programa se ejecuta, o pueden utilizar el aspecto estándar de Java que proporciona una interfaz de usuario idéntica, a través de todas las plataformas de computadoras. Explicamos el paquete **`javax.swing`**, el cual proporciona componentes GUI especialmente poderosos. El capítulo ilustra los principios de diseño de la GUI, la jerarquía **`javax.swing`**, etiquetas, botones de comando, campos de texto, áreas de texto, cuadros combinados, casillas de verificación, paneles, paneles desplegables, paneles a la medida, manejo eventos del ratón, ventanas, menús, y el uso de tres de los administradores más sencillos de diseño de GUI: **`FlowLayout`**, **`BorderLayout`** y **`GridLayout`**. El capítulo se concentra en el modelo de Java para delegación de eventos para el procesamiento de la GUI. Los ejercicios desafían al estudiante a crear GUIs específicas, a ejercitar diversas características de GUI, a desarrollar programas de dibujo que permitan al usuario dibujar con el ratón y a controlar las fuentes.

**Capítulo 30 —Multimedia en Java: imágenes, animación, audio y video—** Trata sobre las capacidades de Java para hacer aplicaciones de computadora "animadas". Es sorprendente que los estudiantes de los primeros cursos de programación estarán escribiendo aplicaciones con todas estas capacidades. Las posibilidades son interesantes. Los estudiantes ahora acceden (por Internet y mediante tecnología en CD-ROM) a bibliotecas enormes de imágenes gráficas, audio y videos, y pueden "relacionarse" con ellos para formar aplicaciones creativas. Casi todas las nuevas computadoras vienen "equipadas con multimedia". Los estudiantes preparan artículos impresionantes y presentaciones de clase con acceso a diversas librerías de imágenes, dibujos, voces, películas, videos, animaciones y otras cosas similares del dominio público. Cuando la mayoría de nosotros es-

tábamos en nuestros primeros grados, un "artículo" era una colección de caracteres, tal vez escritos a mano, o tal vez a máquina. Un "artículo" puede ser un "gran espectáculo" multimedia. Éste puede mantener su interés, alentar su curiosidad, hacerlo sentir lo que los creadores del artículo sintieron cundo estaban haciendo historia. El multimedia puede hacer que sus laboratorios de ciencias sean mucho más interesantes. Los libros de texto pueden cobrar vida. En lugar de observar la imagen estática de algún fenómeno, usted puede verlo en color, con animación, con sonidos, videos y otros efectos. La gente puede aprender más, ahondar más y experimentar más puntos de vista. Una característica del capítulo es la explicación sobre mapas de imagen que permiten a un programa sentir la presencia del puntero del ratón sobre una región de la imagen, sin hacer clic con el ratón. Presentamos una aplicación de mapa de imagen con código activo, con los iconos creados para nuestros tips de programación correspondientes a *Java Multimedia Cyber Classroom*. Conforme el usuario mueva el puntero del ratón sobre las seis imágenes de los iconos, se desplegará la clase de tip, o una "buena práctica de programación" para los iconos de aprobación, o un "tip de portabilidad" para el icono que muestra un insecto con una maleta, etcétera.

***Parte 4: Apéndices***
Los diversos apéndices proporcionan valioso material de referencia. En el apéndice A, presentamos recursos Web y de Internet para C, C++ y Java; en el B presentamos una lista de recursos Web y de Internet para C99; en el apéndice C presentamos gráficos completos sobre asociatividad y precedencia de operadores en C. C++ y Java; en el D, mostramos el conjunto de códigos de caracteres de ASCII. El apéndice E es un manual completo sobre sistemas numéricos que incluye muchos ejercicios de autoevaluación con sus respuestas. El apéndice F proporciona un panorama sobre las bibliotecas estándar de C y los recursos Web para dichas bibliotecas.

## Reconocimientos

Uno de los mayores placeres al escribir un libro de texto es el de reconocer el esfuerzo de mucha gente, cuyos nombres quizá no aparezcan en la portada, pero cuyo arduo trabajo, cooperación, amistad y comprensión fue crucial para la elaboración de este libro. Mucha gente en Deitel & Associates, Inc. dedicó largas horas a este proyecto.

- Abbey Deitel, Presidenta
- Barbara Deitel, Directora de Finanzas
- Christi Kelsey, Directora de Desarrollo de Negocios
- Jeff Listfield, Desarrollador en Jefe
- Su Zhang, Desarrolladora en Jefe

También queremos agradecer a los participantes del College Internship Program de Deitel & Associates: Mike Oliver, Brian O'Connor y Adam Burke, quienes trabajaron en el paquete de accesorios de este libro.[1] En especial queremos agradecer a Tim Christensen. Tim, estudiante de administración de empresas con área de concentración en ciencias de la computación, en su último año en el Boston College, probó todo el código fuente del libro, añadió comentarios a todo el código en C (capítulos 2 a 14), y actualizó los programas de acuerdo con nuestras convenciones de codificación estándar. Creó el apéndice F (Recursos Web y de Internet para las bibliotecas estándar de C) y también trabajó en el paquete de accesorios del libro.

Somos afortunados por haber trabajado en este proyecto con un talentoso y dedicado equipo de editores profesionales de Prentice Hall. Apreciamos de manera especial el extraordinario esfuerzo de nuestra editora de ciencias de la computación, Kate Hargett y su jefa, nuestra mentora en la edición, Marcia Horton, directora

1. Este programa altamente competitivo (recibimos más de 1000 solicitudes para 11 posiciones para prácticas profesionales) ofrece un número limitado de puestos asalariados a los estudiantes del área de Boston en las carreras de Ciencias de la computación, Tecnologías de la información, Mercadotecnia, Administración e Inglés. Los estudiantes trabajan tiempo completo en nuestras oficinas corporativas en Maynard, Massachusetts durante el verano y (para aquellos que van a la universidad en el área de Boston) medio tiempo durante el periodo académico. También ofrecemos puestos de tiempo completo para prácticas profesionales para estudiantes interesados en dejar un semestre la escuela para ganar experiencia en la industria. Los puestos comunes de tiempo completo están disponibles para los egresados. Para mayor información, contacte a nuestra presidenta en **`abbey.deitel@deitel.com`**, y visite nuestro sitio Web, **`www.deitel.com`**.

editorial de la división de ciencias de la computación e ingeniería de Prentice Hall. Vince O'Brien y Tom Manshreck hicieron un estupendo trabajo con el manejo de la producción del libro. Sarah Parker manejó la publicación del amplio paquete de accesorios del libro.

Queremos reconocer el esfuerzo de nuestros revisores de la *cuarta edición*, y dar un agradecimiento especial a Carole Snyder de Prentice Hall, quien condujo este extraordinario esfuerzo de revisión. [Observe que las dos primeras ediciones de *Cómo programar en C* incluyeron sólo a C y C++; Java se añadió hasta la *tercera edición*.]

- Rex Jaeschke (Consultor independiente; presidente del ANSI C Committee)
- John Benito (Representante del grupo de trabajo de ISO que es responsable del lenguaje de programación C)
- Deena Engel (New York University)
- Geb Thomas (University of Iowa)
- Jim Brzowski (University of Massachusetts — Lowell)

Queremos reconocer nuevamente el esfuerzo de nuestros revisores anteriores (algunos de la primera edición, algunos de la segunda, algunos otros de la tercera, y algunos de todas); los puestos estaban vigentes al momento de la revisión:

- Rex Jaeschke (Consultor independiente; presidente del ANSI C Committee)
- Randy Meyers (NetCom; miembro del ANSI C Committee; presidente del ANSI C++ Committee)
- Simon North (Synopsis, Autor de XML)
- Fred Tydeman (Consultor)
- Kevin Wayne (Princeton University)
- Eugene Katzin (Montgomery College)
- Sam Harbison (Texas Instruments, Autor de PH)
- Chuck Allison (Consultor de Tydeman)
- Catherine Dwyer (Pace University)
- Glen Lancaster (DePaul University)
- David Falconer (California State University en Fullerton)
- David Finkel (Worcester Polytechnic)
- H. E. Dunsmore (Purdue University)
- Jim Schmolze (Tufts University)
- Gene Spafford (Purdue University)
- Clovis Tondo (IBM Corporation y profesor eventual de Nova University)
- Jeffrey Esakov (University of Pennsylvania)
- Tom Slezak (University of California, Lawrence Livermore National Laboratory)
- Gary A. Wilson (Gary A Wilson & Associates y University of California Berkeley Extension)
- Mike Kogan (IBM Corporation; arquitecto en jefe de 32-bit OS/2 2.0)
- Don Kostuch (retirado de IBM Corporation; instructor internacional de C, C++ y de programación orientada a objetos)
- Ed Lieblein (Nova University)
- John Carroll (San Diego State University)
- Alan Filipski (Arizona State University)
- Greg Hidley (University of California, San Diego)
- Daniel Hirschberg (University of California, Irvine)
- Jack Tan (University of Houston)
- Richard Alpert (Boston University)
- Eric Bloom (Bentley College)

Estos revisores examinaron cada aspecto del libro e hicieron incontables sugerencias para mejorar la precisión e integridad de esta presentación.

***Cómo ponerse en contacto con Deitel & Associates***

Agradeceremos mucho sus comentarios, críticas, correcciones y sugerencias para mejorar este libro.

Remita sus preguntas respecto al lenguaje C, C++ y Java a:

**deitel@deitel.com**

le responderemos oportunamente.

***Erratas***

Anunciaremos todas las erratas de la *cuarta edición* en **www.deitel.com**.

Bienvenido al excitante mundo de la programación por procedimientos en C, a la programación general, a la basada en objetos y a la orientada a objetos en C++, y a la programación de gráficos, de la interfaz gráfica de usuario, multimedia y dirigida por eventos en Java. Esperamos sinceramente que disfrute su aprendizaje con este libro.

*Dr. Harvey M. Deitel*
*Paul J. Deitel*

## Acerca de los autores

**Dr. Harvey M. Deitel**. Presidente y director en jefe de estrategia (SCO) de Deitel & Associates, Inc. Tiene 42 años de experiencia en el campo de la computación, esto incluye un amplio trabajo académico y en la industria. El Dr. Deitel tiene una licenciatura y una maestría por el Massachusetts Institute of Technology, y un doctorado por la Boston University. Trabajó en los primeros proyectos de sistemas operativos de memoria virtual en IBM y el MIT que desarrollaron técnicas que en la actualidad están ampliamente implementadas en sistemas tales como UNIX, Linux y Windows XP. Tiene 20 años de experiencia como profesor universitario, la cual incluye un puesto vitalicio y el haber servido como presidente del departamento de Ciencias de la computación en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Él y Paul son coautores de varias docenas de libros y paquetes multimedia, y están escribiendo muchos más. Los textos del Dr. Deitel se han ganado el reconocimiento internacional y han sido traducidos al Japonés, Ruso, Español, Chino tradicional, Chino simplificado, Coreano, Francés, Polaco, Italiano, Portugués, Griego, Urdú y Turco. El Dr. Deitel ha impartido seminarios profesionales para grandes empresas, organizaciones gubernamentales y diversos sectores del ejército.

**Paul J. Deitel**. CEO y director técnico en jefe de Deitel & Associates, Inc., es egresado del Sloan School of Management del Massachusetts Institute of Technology, en donde estudió Tecnología de la Información. A través de Deitel & Associates, Inc., ha impartido cursos de C, C++, Java, Internet y sobre World Wide Web a clientes de la industria, como IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA (en el centro espacial Kennedy) National Severe Storm Laboratory, Compaq, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, Cable-Data Systems, y muchas otras. Ha ofrecido conferencias de C++ y Java para la Boston Chapter de la Association for Computing Machinery, y ha impartido cursos sobre Java para comunicación satelital a través de una empresa de cooperación entre Deitel & Associates, Prentice Hall y Technology Education Network. Él y su padre, el Dr. Harvey M. Deitel, son los autores de los libros de Ciencias de la computación con más ventas en el mundo.

## Acerca de DEITEL & Associates, Inc.

Deitel & Associates, Inc., es una empresa reconocida a nivel mundial dedicada al entrenamiento y creación de contenidos especializados en educación para tecnología de software para Internet/World Wide Web, tecnología de software para comercio electrónico (e-business/e-commerce), tecnología de objetos y lenguajes de programación. La empresa proporciona cursos guiados sobre programación en Internet y World Wide Web, programación inalámbrica para Internet, tecnología de objetos, y lenguajes y plataformas importantes de programación como C, C++, Visual C++®.NET, Visual Basic®.NET, C#, Java, Java avanzado, XML, Perl, Phyton y otros. Los fundadores de Deitel & Associates, Inc., son el Dr. Harvey M. Deitel y Paul J. Deitel. Entre sus clientes se encuentran muchas de las empresas de cómputo más grandes del mundo, agencias gubernamentales, sectores del ejército y organizaciones de negocios. A lo largo de sus 27 años de sociedad editorial con Prentice Hall,

Deitel & Associates, Inc., ha publicado libros de texto de vanguardia sobre programación, libros profesionales, multimedia interactiva en CD como los *Cyber Classrooms*, *Complete Training Courses*, cursos basados en Web y un curso de administración de sistemas con contenido electrónico para CMSs populares como WebCT™, Blackboard™ y CourseCompass℠. Puede contactar a Deitel & Associates, Inc., y a los autores mediante correo electrónico en:

**deitel@deitel.com**

Para conocer más sobre Deitel & Associates, Inc., sus publicaciones y su currículo corporativo mundial visite:

**www.deitel.com**



# Introducción a las computadoras, a Internet y a la World Wide Web




## Objetivos

- Comprender los conceptos básicos acerca de las computadoras.
- Familiarizarse con diferentes tipos de lenguajes de programación.
- Familiarizarse con la historia del lenguaje de programación C.
- Conocer la biblioteca estándar de C.
- Comprender los elementos de un entorno de desarrollo típico de C.
- Apreciar por qué es apropiado aprender C como primer curso de programación.
- Apreciar por qué C proporciona los fundamentos para el estudio de otros lenguajes de programación en general, y en particular para C++, Java y C#.
- Familiarizarse con la historia de Internet y de la World Wide Web.

*Las cosas siempre son mejores al principio.*
Blaise Pascal

*Las grandes ideas requieren un lenguaje grande*.
Aristófanes

*Nuestra vida siempre es malgastada por el detalle… simplificar, simplificar*.
Henry Thoreau

## Plan general



## 1.1 Introducción

¡Bienvenidos a C, C++ y Java! Hemos trabajado duro para crear lo que creemos será una experiencia educativa informativa y entretenida para usted. Este libro es único entre otros libros de texto de C porque:

- Es apropiado para gente con orientación técnica que cuente con poca o nada de experiencia en programación.
- Es adecuado para programadores experimentados que deseen conocer más profundamente el lenguaje.

¿Cómo puede un libro ser atractivo para ambos grupos? La respuesta es que la parte central del libro pone énfasis en la *claridad* de los programas, a través de las técnicas comprobadas de *programación estructurada*. Los principiantes aprenden a programar bien desde el principio. Hemos intentado escribir de manera clara y directa. El libro contiene ilustraciones en abundancia. Quizá lo más importante sea que el libro contiene cientos de programas completos, los cuales muestran los resultados que arrojan cuando se ejecutan en una computadora. Nosotros llamamos a esto "el método del código activo". Todos estos programas de ejemplo se encuentran en el CD-ROM que acompaña a este libro; también puede descargar los originales desde nuestra página Web `www.deitel.com`.

Los primeros cuatro capítulos presentan los fundamentos de las computación, de la programación de computadoras y del lenguaje de programación C. Los principiantes que han tomado nuestros cursos nos han dicho que el material que presentamos en estos capítulos contiene una base sólida para un tratamiento más profundo de C en los capítulos restantes. Los programadores experimentados por lo general leen rápidamente los cuatro primeros capítulos, y encuentran que el tratamiento de C en los capítulos 5 a 14 es más riguroso y desafiante. En particular, aprecian el tratamiento profundo de apuntadores, cadenas, archivos y estructuras de datos de los capítulos restantes.

Muchos programadores experimentados aprecian el tratamiento de la programación estructurada. A menudo han programado en un lenguaje estructurado como Pascal, pero debido a que no recibieron una introducción formal a la programación estructurada, no escriben con el mejor código posible. Conforme aprenden C con este libro, mejoran su estilo de programación. De manera que, si es usted un programador principiante o experimentado, aquí le presentamos mucho material para informarlo, entretenerlo y desafiarlo.

La mayoría de la gente está familiarizada con las cosas excitantes que se pueden hacer con una computadora. Mediante este libro de texto, usted aprenderá a programar las computadoras para que hagan dichas cosas. El *software* (es decir, las instrucciones que usted escribe para ordenar a la computadora que realice *acciones* y tome *decisiones*) es quien controla a las computadoras (a menudo llamadas *hardware*). Este libro presenta una introducción a la programación en C, el cual se estandarizó en 1989 en Estados Unidos a través del *American National Standards Institute (ANSI)*, y a nivel mundial a través de los esfuerzos de la *International Standards Organization (ISO)*.

El uso de las computadoras ha invadido casi cualquier campo de trabajo. En una era de constantes aumentos de costos, los de cómputo han disminuido de manera dramática debido al rápido desarrollo de la tecnología de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares hace dos décadas, ahora se colocan en las superficies de pequeños chips de silicio, más pequeños que una uña y con un costo de quizá unos cuántos dólares cada uno. De manera irónica, el silicio es uno de los materiales más abundantes en el planeta (es uno de los ingredientes de la tierra común). La tecnología de los chips de silicio ha vuelto tan económica a la computación que cientos de miles de computadoras de uso común se encuentran actualmente ayudando a la gente en las empresas, en la industria, en el gobierno y en sus vidas personales. Dicho número podría duplicarse en unos cuantos años.

En la actualidad, C++ y Java (lenguajes de programación orientados a objetos, basados en C) reciben tanta atención, que en los capítulos 15 a 23 incluimos una completa introducción a la programación orientada a objetos en C++, y en los capítulos 24 a 30 una completa introducción a la programación orientada a objetos en Java. En el mercado de los lenguajes de programación, muchos fabricantes combinan C y C++ en un solo producto, en lugar de ofrecerlos por separado. Esto les brinda a los usuarios la posibilidad de continuar programando en C, y de manera gradual migrar a C++ cuando sea apropiado. Todo el software que necesita para desarrollar y ejecutar los programas en C, C++ y Java correspondientes a este libro se encuentra disponible ya sea en el CD que lo acompaña, o lo puede descargar de manera gratuita desde Internet. (Consulte el Prefacio.)

Está a punto de comenzar una ruta de desafíos y recompensas. Mientras tanto, si desea comunicarse con nosotros, envíenos un correo electrónico a:

**deitel@deitel.com**

o explore nuestra página Web en:

**www.deitel.com**

Le responderemos pronto. Esperamos que disfrute su aprendizaje en C, C++ y Java.

## 1.2 ¿Qué es una computadora?

Una *computadora* es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones (incluso miles de millones) de veces más rápidas que los humanos. Por ejemplo, muchas de las computadoras personales actuales pueden realizar miles de millones de sumas por segundo. Una persona con una calculadora podría requerir toda una vida para completar el mismo número de operaciones que una poderosa computadora realiza en un segundo. (Puntos a considerar: ¿cómo sabría que una persona realizó los cálculos de manera correcta? ¿Cómo sabría que la computadora lo hizo de manera correcta?) ¡Las *supercomputadoras* actuales más rápidas pueden realizar miles de millones de sumas por segundo! ¡Y en los laboratorios de investigación se encuentran otras que pueden realizar billones de instrucciones por segundo!

Las computadoras procesan los *datos* bajo el control de conjuntos de instrucciones llamadas *programas de cómputo*. Estos programas de cómputo guían a la computadora a través de conjuntos ordenados de acciones especificados por personas llamadas *programadores de computadoras*.

Una computadora está compuesta por varios dispositivos (tales como el teclado, el monitor, el “ratón”, discos, memoria, DVD, CD-ROM y unidades de procesamiento) conocidos como *hardware*. A los programas de

cómputo que se ejecutan dentro de una computadora se les denomina *software*. En años recientes, los costos de las piezas de hardware han disminuido de manera espectacular, al punto de que las computadoras personales se han convertido en artículos domésticos. Por desgracia, los costos para el desarrollo de programas se incrementan de manera constante conforme los programadores desarrollan aplicaciones más complejas y poderosas, sin que exista una mejora significativa en la tecnología para el desarrollo de software. En este libro aprenderá métodos comprobados para el desarrollo de software que están ayudando a las empresas a controlar e incluso a reducir sus costos (programación estructurada, mejoramiento paso a paso, uso de funciones, programación basada en objetos, programación orientada a objetos, diseño orientado a objetos y programación genérica).

## 1.3 Organización de computadoras

Independientemente de la apariencia física, casi siempre podemos representar a las computadoras mediante seis *unidades* o secciones *lógicas*:

1. *Unidad de entrada*. Ésta es la sección "receptora" de la computadora. Obtiene información (datos y programas de cómputo) desde varios *dispositivos de entrada* y pone esta información a disposición de las otras unidades para que la información pueda procesarse. La mayor parte de la información se introduce a través del teclado y el ratón. La información también puede introducirse hablando con su computadora, digitalizando las imágenes y mediante la recepción de información desde una red, como Internet.

2. *Unidad de salida*. Ésta es la sección de "embarque" de la computadora. Toma información que ya ha sido procesada por la computadora y la coloca en los diferentes *dispositivos de salida*, para que la información esté disponible fuera de la computadora. La mayor parte de la información de salida se despliega en el monitor, se imprime en papel, o se utiliza para controlar otros dispositivos. Las computadoras también pueden dar salida a su información a través de redes, tales como Internet.

3. *Unidad de memoria.* Ésta sección funciona en la computadora como un "almacén" de acceso rápido, pero con una capacidad relativamente baja. Ésta retiene la información que se introduce a través de la unidad de entrada, de manera que la información pueda estar disponible de manera inmediata para procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada, hasta que la unidad de salida pueda colocarla en los dispositivos de salida. Con frecuencia, a la unidad de memoria se le llama *memoria* o *memoria principal*.

4. *Unidad aritmética y lógica (ALU).* Ésta es la sección de "manufactura" de la computadora. Es la responsable de realizar cálculos tales como suma, resta, multiplicación y división. Contiene los mecanismos de decisión que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no.

5. *Unidad central de procesamiento (CPU).* Ésta es la sección "administrativa" de la computadora; es quien coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, le indica a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y le indica a la unidad de salida cuándo enviar la información desde la unidad de memoria hacia ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples unidades de procesamiento y, por lo tanto, pueden realizar múltiples operaciones de manera simultánea (a estas computadoras se les conoce como *multiprocesadoras*).

6. *Unidad secundaria de almacenamiento*. Éste es el "almacén" de alta capacidad y de larga duración de la computadora. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan dentro de dispositivos de almacenamiento secundario (tales como discos) hasta que son requeridos de nuevo, posiblemente horas, días, meses o incluso años después. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal.

## 1.4 Evolución de los sistemas operativos

Las primeras computadoras eran capaces de realizar solamente una *tarea* o *trabajo* a la vez. A esta forma de operación de la computadora a menudo se le conoce como *procesamiento por lotes* (batch) de un solo usuario. La computadora ejecuta un solo programa a la vez, mientras procesa los datos en grupos o *lotes*. En estos primeros sistemas, los usuarios generalmente asignaban sus trabajos a un centro de cómputo que los introducía en paquetes de tarjetas perforadas. A menudo tenían que esperar horas, e incluso días, antes de que sus resultados impresos regresaran a sus escritorios.

Los sistemas de software denominados *sistemas operativos* fueron desarrollados para hacer más fácil el uso de la computadora. Los primeros sistemas operativos administraban la suave transición entre tareas. Esto minimizó el tiempo necesario para que los operadores de computadoras pasaran de una tarea a otra, y por consiguiente incrementó la cantidad de trabajo, o el *flujo de datos*, que las computadoras podían procesar.

Conforme las computadoras se volvieron más poderosas, se hizo evidente que un proceso por lotes para un solo usuario rara vez aprovechaba los recursos de la computadora de manera eficiente, debido al tiempo que se malgastaba esperando a que los lentos dispositivos de entrada/salida completaran sus tareas. Se pensó que era posible realizar muchas tareas o trabajos que podrían *compartir* los recursos de la computadora y lograr un uso más eficiente de ésta. A esto se le conoce como *multiprogramación*. La multiprogramación significa la operación "simultánea" de muchas tareas dentro de la computadora (la computadora comparte sus recursos entre los trabajos que compiten por su atención). En los primeros sistemas operativos con multiprogramación, los usuarios aún tenían que enviar sus trabajos mediante paquetes de tarjetas perforadas y esperar horas o días por sus resultados.

En la década de los sesenta, muchos grupos de la industria y de las universidades marcaron los rumbos de los sistemas operativos de *tiempo compartido*. El tiempo compartido es un caso especial de la multiprogramación, en el cual, los usuarios acceden a la computadora a través de *terminales*; por lo general, dispositivos compuestos por un teclado y un monitor. En un típico sistema de cómputo de tiempo compartido puede haber docenas o incluso cientos de usuarios compartiendo la computadora al mismo tiempo. La computadora en realidad no ejecuta los procesos de todos los usuarios a la vez. Ésta hace el trabajo tan rápido que puede proporcionar el servicio a cada usuario varias veces por segundo. Así, los programas de los usuarios *aparentemente* se ejecutan de manera simultánea. Una ventaja del tiempo compartido es que el usuario recibe respuestas casi inmediatas a las peticiones, en vez de tener que esperar los resultados durante largos periodos, como en los comienzos de la computación.

## 1.5 Computación personal, distribuida y cliente-servidor

En 1977, Apple Computers popularizó el fenómeno de la *computación personal*. Al principio era el sueño de todo aficionado. Las computadoras se hicieron lo suficientemente económicas para que la gente las pudiera adquirir para su uso personal o para negocios. En 1981, IBM, el vendedor de computadoras más grande del mundo, introdujo la PC de IBM. Literalmente, de la noche a la mañana, la computación personal se posicionó en las empresas, en la industria y en las instituciones gubernamentales.

Estas computadoras eran unidades "independientes" (la gente hacía su trabajo en su propia máquina y transportaba sus discos de un lado a otro para compartir información). Aunque las primeras computadoras personales no eran lo suficientemente poderosas para compartir el tiempo entre muchos usuarios, estas máquinas podían interconectarse entre sí mediante redes, algunas veces mediante líneas telefónicas y otras mediante *redes de área local (LANs)* dentro de la empresa. Esto derivó en el fenómeno denominado *computación distribuida*, en el que la computación de la empresa, en vez de llevarse a cabo dentro de un centro de cómputo, se distribuye a través de redes a los sitios en donde se realiza el trabajo de la empresa. Las computadoras personales eran lo suficientemente poderosas para manejar los requerimientos de cómputo de usuarios individuales, y para manejar de manera electrónica las tareas básicas de comunicación que involucraba la transferencia de información entre una computadora y otra.

Las computadoras personales actuales son tan poderosas como las máquinas de un millón de dólares de hace apenas una década. Las máquinas de escritorio más poderosas (denominadas *estaciones de trabajo*) proporcionan al usuario enormes capacidades. La información se comparte de manera muy sencilla a través de redes de computadoras, en donde algunas computadoras denominadas *servidores de archivos* ofrecen un lugar

común de almacenamiento para programas y datos que pueden ser utilizados por computadoras *cliente* distribuidas a través de la red; de ahí el término de computación *cliente-servidor*. C, C++ y Java son lenguajes de programación ampliamente utilizados para crear software para sistemas operativos, para redes de computadoras y para aplicaciones distribuidas cliente-servidor. Los sistemas operativos más populares tales como UNIX, Linux, OS X de Mac y Windows proporcionan el tipo de capacidades que explicamos en esta sección.

## 1.6 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de estos lenguajes los comprende directamente la computadora, mientras que otros requieren pasos intermedios de *traducción*. En la actualidad se utilizan cientos de lenguajes de computación, los cuales se dividen en tres tipos generales:

1. Lenguajes máquina.
2. Lenguajes ensambladores.
3. Lenguajes de alto nivel.

Cualquier computadora puede entender de manera directa sólo su propio *lenguaje máquina*. El lenguaje máquina es el "lenguaje natural" de una computadora en particular, y está definido por el diseño del hardware de dicha computadora. Por lo general, los lenguajes máquina consisten en cadenas de números [que finalmente se reducen a unos (1) y ceros (0)] que instruyen a las computadoras para realizar sus operaciones más elementales, una por una. Los lenguajes máquina son *dependientes de la máquina*, es decir, un lenguaje máquina en particular puede utilizarse solamente en un tipo de computadora. Los lenguajes máquina son difíciles de comprender para los humanos, como podrá ver en el programa de lenguaje máquina de la siguiente sección, el cual suma el pago de horas extras a un sueldo base y lo almacena en un sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. En lugar de utilizar las cadenas de números que las computadoras podían entender de manera directa, los programadores comenzaron a utilizar abreviaturas del inglés para representar las operaciones básicas de la computadora. Estas abreviaturas del inglés formaron la base de los *lenguajes ensambladores*. Los *programas traductores* llamados *ensambladores* se desarrollaron para convertir programas en lenguaje ensamblador a lenguaje máquina a la velocidad de la computadora. La siguiente sección muestra un programa en lenguaje ensamblador que también suma el pago por horas extras a un sueldo base y almacena el resultado en un sueldo bruto, pero de manera más clara que su equivalente en lenguaje máquina:

```
LOAD    SUELDOBASE
ADD     SUELDOEXTRA
STORE   SUELDOBRUTO
```

Aunque dicho código es más claro para los humanos, será incomprensible para las computadoras, hasta que los ensambladores lo traduzcan al lenguaje máquina.

El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero éstos aún requerían muchas instrucciones para llevar a cabo las tares más sencillas. Para acelerar el proceso de programación, se desarrollaron los *leguajes de alto nivel*, en los que las instrucciones individuales llevan a cabo tareas importantes. A los programas traductores que convierten programas escritos en lenguajes de alto nivel a lenguaje máquina, se les llama *compiladores*. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que se parecen mucho al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener una instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra
```

Obviamente, los lenguajes de alto nivel son mucho más recomendables, desde el punto de vista del programador, que los lenguajes máquina y ensamblador. C, C++ y Java son los lenguajes de alto nivel más poderosos, y más ampliamente utilizados.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje máquina puede tardar un tiempo considerable. Los programas *intérpretes* se desarrollaron para que pudieran ejecutar programas de alto nivel sin necesidad de compilar dichos programas a lenguaje máquina. Aunque la ejecución de los programas compilados es más rápida que los programas interpretados, los intérpretes son populares en ambientes de desarrollo de programas, en los cuales los programas se recompilan de manera frecuente conforme se adicionan nuevas características y se corrigen los errores. Una vez que se desarrolla un programa, una versión compilada puede ejecutarse de manera más eficiente.

## 1.7 FORTRAN, COBOL, Pascal y Ada

Se han desarrollado cientos de lenguajes de alto nivel, pero sólo algunos han logrado tener gran aceptación. En la década de los cincuenta, IBM Corporation desarrolló *FORTRAN* (FORmula TRANslator, traductor de formulas) para que se utilizara en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. Actualmente, FORTRAN se utiliza ampliamente, en especial en aplicaciones de ingeniería.

*COBOL* (COmmon Business Oriented Language, lenguaje común orientado a los negocios) fue desarrollado en 1959 por fabricantes de computadoras, el gobierno y los usuarios de computadoras en la industria. COBOL se utiliza para aplicaciones comerciales que requieren una manipulación precisa y eficiente de grandes cantidades de datos. Una considerable cantidad de software de negocios se encuentra todavía programada en COBOL.

Durante la década de los sesenta, muchas de las grandes iniciativas para desarrollo de software encontraron severas dificultades. Los itinerarios de software generalmente se retrasaban, los costos rebasaban en gran medida los presupuestos, y los productos terminados no eran confiables. La gente comenzó a darse cuenta de que el desarrollo de software era una actividad mucho más compleja de lo que habían imaginado. Las actividades de investigación durante esta década dieron como resultado la evolución de la *programación estructurada* (un método disciplinado para escribir programas más claros, fáciles de corregir, y más fáciles de modificar).

Uno de los resultados más tangibles de esta investigación fue el desarrollo del lenguaje de programación Pascal por el profesor Niklaus Wirth, en 1971. Pascal, cuyo nombre se debe al aniversario de los setecientos años del nacimiento del filósofo y matemático Blaise Pascal, fue diseñado para la enseñanza de la programación estructurada en ambientes académicos, y de inmediato se convirtió en el lenguaje de programación favorito en varias universidades. Desafortunadamente, el lenguaje carecía de muchas de las características necesarias para poder utilizarlo en aplicaciones comerciales, industriales y gubernamentales, por lo que no ha sido muy aceptado en estos ambientes.

El lenguaje de programación *Ada* fue desarrollado bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DoD) durante la década de los setenta y principios de la década de los ochenta. Cientos de lenguajes se utilizaron para producir los sistemas de software de comando y control masivo del departamento de defensa. El departamento de defensa quería un lenguaje único que pudiera cubrir la mayoría de sus necesidades. El nombre del lenguaje es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A Lady Lovelace se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de 1800 (para la Máquina Analítica, un dispositivo de cómputo creado por Charles Babbage). Una de las características importantes de Ada se conoce como *multitareas*; esto permite a los programadores especificar que ocurrirán varias tareas en paralelo. Algunos de los lenguajes de alto nivel más populares que hemos explicado (incluyendo C y C++) generalmente permiten al programador escribir programas que realizan solo una actividad a la vez. Java, mediante una técnica denominada subprocesamiento múltiple, permite a los programadores escribir programas con actividades en paralelo.

## 1.8 Historia de C

C evolucionó de dos lenguajes de programación anteriores, BCPL y B. En 1967, Martin Richards desarrolló BCPL como un lenguaje para escribir software para sistemas operativos y compiladores. Ken Thompson, en su lenguaje B, modeló muchas de las características de C, luego del desarrollo de su contraparte en BCPL y, en 1970, utilizó B para crear las primeras versiones del sistema operativo UNIX en los laboratorios Bell, sobre una computadora DEC PDP-7. Tanto BCPL como B eran lenguajes “sin tipo” (cada dato ocupaba una “palabra” en memoria y, por ejemplo, el trabajo de procesar un elemento como un número completo o un número real, era responsabilidad del programador).

El lenguaje C evolucionó a partir de B; dicha evolución estuvo a cargo de Dennis Ritchie en los laboratorios Bell y, en 1972, se implementó en una computadora DEC PDP-11. C utiliza muchos conceptos importantes de BCPL y B cuando agrega tipos de datos y otras características. Inicialmente, C se hizo popular como lenguaje de desarrollo para el sistema operativo UNIX. En la actualidad, la mayoría de los sistemas operativos están escritos en C y/o C++. C se encuentra disponible para la mayoría de las computadoras, y es independiente del hardware. Con un diseño cuidadoso, es posible escribir programas en C que sean *portables* para la mayoría de las computadoras.

Para fines de la década de los setenta, C evolucionó a lo que ahora se conoce como "C tradicional", "C clásico", o "C de Kernigham y Ritchie". La publicación que en 1978 Prentice Hall hiciera del libro de Kernigham y Ritchie, *El lenguaje de programación C*, atrajo mucho la atención de la gente a dicho lenguaje. Esta publicación se convirtió en uno de los textos de computación más exitoso de todos los tiempos.

La amplia utilización de C para distintos tipos de computadoras (en ocasiones llamadas *plataformas de hardware*) ocasionó, por desgracia, muchas variantes. Éstas eran similares, pero a menudo incompatibles, lo que se volvió un problema serio para los desarrolladores que necesitaban escribir programas que se pudieran ejecutar en distintas plataformas. Entonces se hizo evidente la necesidad de una versión estándar de C. En 1983, se creó el comité técnico X3J11 bajo la supervisión del American National Standards Comittee on Computers and Information Processing (X3), para "proporcionar una definición clara del lenguaje e independiente de la computadora". En 1989, el estándar fue aprobado; éste estándar se actualizó en 1999. Al documento del estándar se le conoce como *INCITS/ISO/IEC 9899-1999*. Usted puede solicitar una copia de este documento a la American National Standards Institute (`www.ansi.org`) en `webstore.ansi.org/ansidocstore`.



**Tip de portabilidad 1.1**

*Debido a que C es un lenguaje ampliamente disponible, independiente de la plataforma, y estandarizado, las aplicaciones escritas en C a menudo pueden ejecutarse sobre un amplio rango de sistemas de cómputo con muy pocas o ninguna modificación.*

[*Nota:* Incluiremos muchos de estos *Tips de portabilidad* para resaltar técnicas que le ayudarán a escribir programas que se puedan ejecutar, con poca o ninguna modificación, en una variedad de computadoras. También resaltaremos las *Buenas prácticas de programación* (prácticas que le pueden ayudar a escribir programas más claros, comprensibles, fáciles de mantener y fáciles de probar y depurar, esto es, eliminar errores), *Errores comunes de programación* (errores de los que se debe cuidar, de manera que no los cometa en sus programas), *Tips de rendimiento* (técnicas que le ayudarán a escribir programas que se ejecuten más rápido y que utilicen menos memoria), *Tips para prevenir errores* (técnicas que le ayudarán a eliminar errores de sus programas, y lo más importante, técnicas que le ayudarán a escribir programas libres de errores desde el principio), y *Observaciones de ingeniería de software* (conceptos que afectan y mejoran la arquitectura general y la calidad de un sistema de software, y en particular, de un gran número de sistemas). Muchas de éstas técnicas y prácticas son solamente guías; sin duda, usted deberá desarrollar su propio estilo de programación.]

## 1.9 La biblioteca estándar de C

Como verá en el capítulo 5, los programas en C constan de módulos o piezas llamadas *funciones*. Usted puede programar todas las funciones que necesite para formar un programa en C, pero la mayoría de los programadores aprovechan la rica colección de funciones existentes dentro de la llamada *Biblioteca Estándar de C*. Además, en realidad existen dos claves para aprender a programar en C. La primera es aprender el propio lenguaje C, y la segunda es aprender la manera de utilizar las funciones de la biblioteca estándar. A través del texto, explicaremos muchas de estas funciones. El libro de P. J. Plauger, *The Standard C Library,* es una lectura obligada para aquellos programadores que necesitan comprender profundamente las funciones de la biblioteca, cómo implementarlas y cómo escribir código que sea portable.

El texto fomenta un *método de construcción por bloques* para crear programas. Evite reinventar la rueda. Utilice piezas existentes, a esto se le denomina *reutilización de software*, y es clave para el campo de la programación orientada a objetos, como veremos en los capítulos 15 a 30. Cuando programe en C, por lo general utilizará los siguientes bloques de construcción:

- Funciones de la biblioteca estándar de C.
- Funciones creadas por usted mismo.

• Funciones creadas por otras personas y disponibles para usted.

La ventaja de crear sus propias funciones es que conocerá con exactitud cómo funcionan. Será capaz de examinar el código en C. La desventaja es el tiempo y el esfuerzo que involucra el diseño y el desarrollo de las nuevas funciones.

Si utiliza funciones existentes, puede evitar la reinvención de la rueda. En el caso de las funciones del estándar ANSI, usted sabe que están escritas con mucho cuidado, y sabe que, debido a que utiliza funciones que se encuentran disponibles virtualmente en todas las implementaciones de ANSI C, sus programas tendrán grandes posibilidades de ser portables.

**Tip de rendimiento 1.1**

*Utilizar funciones de la biblioteca estándar de ANSI, en lugar de escribir sus propias funciones similares, puede mejorar el rendimiento del programa, debido a que estas funciones están escritas cuidadosamente para una ejecución eficiente.*



**Tip de portabilidad 1.2**

*Utilizar funciones de la biblioteca estándar de ANSI, en lugar de escribir sus propias funciones similares, puede mejorar la portabilidad, debido a que estas funciones se utilizan virtualmente en cualquier implementación del C de ANSI.*

## 1.10 C++

C++ es un C mejorado, desarrollado por Bjarne Stroustrup en los laboratorios Bell. C++ proporciona un conjunto de características que "pulen" al lenguaje C; sin embargo, lo más importante es que proporciona capacidades para una *programación orientada a objetos*. C++ se ha convertido en el lenguaje dominante en la industria y en las universidades.

Los *objetos* son, esencialmente, *componentes* reutilizables de software que modelan elementos reales. Una revolución se está gestando en la comunidad del software. Escribir software rápida, correcta y económicamente es aún una meta escurridiza, en una época en la que la demanda de nuevo y más poderoso software se encuentra a la alza.

Los desarrolladores de software están descubriendo que utilizar una metodología de diseño e implementación modular y orientada a objetos puede hacer más productivos a los grupos de desarrollo de software, que mediante las populares técnicas de programación anteriores.

Muchas personas piensan que la mejor estrategia educativa actual es dominar C, y posteriormente estudiar C++. Por lo tanto, en los capítulos 15 a 23 del presente libro, presentaremos una explicación resumida de C++, la cual extrajimos de nuestro libro *C++ Cómo programar*. Esperamos que lo encuentre valioso y que lo motive para que al terminar este texto estudie C++.

## 1.11 Java

Mucha gente cree que el próximo campo importante en el que los microprocesadores tendrán un impacto profundo es en los dispositivos electrónicos inteligentes para uso doméstico. Al aceptar esto, Sun Microsystems patrocinó, en 1991, un proyecto de investigación de la empresa denominado Green. El proyecto desembocó en el desarrollo de un lenguaje basado en C y C++, al cual, James Gosling llamó Oak, debido a un roble que tenía a la vista desde su ventana en las oficinas de Sun. Posteriormente se descubrió que ya existía un lenguaje de programación con el mismo nombre. Cuando un grupo de gente de Sun visitó una cafetería local, sugirieron el nombre *Java* (una variedad de café), y así se quedó.

Sin embargo, el proyecto Green tuvo algunas dificultades. El mercado para los dispositivos electrónicos inteligentes de uso doméstico no se desarrollaba tan rápido como Sun había anticipado. Peor aún, un contrato importante por el que Sun había competido, se le otorgó a otra empresa. De manera que el proyecto corría peligro de ser cancelado. Pero para su buena fortuna, la popularidad de la World Wide Web explotó en 1993, y la gente de Sun se dio cuenta de inmediato del potencial de Java para crear *contenido dinámico* para páginas Web.

Sun anunció formalmente a Java en una exposición profesional que tuvo lugar en mayo de 1995. De inmediato, Java generó interés dentro de la comunidad de negocios debido a la fenomenal explosión de la World Wide Web. En la actualidad, Java se utiliza para crear páginas Web con contenido dinámico e interactivo, para

desarrollar aplicaciones a gran escala, para aumentar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en los navegadores Web), para proporcionar aplicaciones para dispositivos domésticos (como teléfonos celulares, localizadores y asistentes digitales personales), y más.

En 1995, estábamos siguiendo el desarrollo de Java. En noviembre de 1995, asistimos a una conferencia sobre Internet que tuvo lugar en Boston. Un representante de Sun Microsystems dio una animada presentación sobre Java. Mientras la plática se llevaba a cabo, se hizo evidente para nosotros que Java tendría un papel importante en el desarrollo de páginas Web interactivas y con multimedia. Sin embargo, de inmediato vimos un potencial mucho mayor para el lenguaje. Vimos a Java como un magnífico lenguaje para enseñar a los estudiantes de primer año de programación los fundamentos de la computación con gráficos, con imágenes, animación, audio, video, con bases de datos, redes, con subprocesamiento múltiple y de colaboración.

Los capítulos 24 a 30 de este libro presentan una introducción detallada a los gráficos en Java, la programación de interfaces gráficas de usuario (GUI), programación multimedia y programación basada en eventos. Este material está cuidadosamente condensado y extraído de nuestro libro *Java, Cómo programar*. Esperamos que usted encuentre este material valioso y que lo motive para que continúe con un estudio más profundo de Java.

Además de su prominencia para desarrollar aplicaciones para Internet e intranets, Java se ha convertido en el lenguaje a elegir para implementar software para dispositivos que se comunican a través de una red (tales como teléfonos celulares, localizadores y asistentes electrónicos personales) ¡No se sorprenda si su nuevo equipo de sonido y otros dispositivos de su hogar pueden conectarse entre sí mediante el uso de la tecnología Java¡

## 1.12 BASIC, Visual Basic, Visual C++, C# y .NET

El lenguaje de programación BASIC (Beginner´s All-Purpose Symbolic Instruction Code) fue desarrollado a mediados de la década de los sesenta por los profesores del Darmouth College John Kemeny y Thomas Kurtz, como un lenguaje para escribir programas sencillos. El propósito principal de BASIC era familiarizar a los principiantes con las técnicas de programación. Visual Basic fue introducido por Microsoft en 1991 para simplificar el proceso de desarrollo de aplicaciones para Windows.

Visual Basic .NET, Visual C++ .NET y C# fueron diseñados para la nueva *plataforma de programación* de Microsoft llamada .NET. Estos tres lenguajes utilizan la poderosa biblioteca de componentes reutilizables de software llamada Framework Class Library (FCL).

De manera comparable a Java, la plataforma .NET permite la distribución de aplicaciones basadas en la Web hacia muchos dispositivos (incluso teléfonos celulares) y computadoras de escritorio. El lenguaje de programación C# fue diseñado de manera específica para la plataforma .NET como el lenguaje que permitiría a los programadores migrar con facilidad hacia .NET. C++, Java y C# tienen todos sus raíces en el lenguaje de programación C.

## 1.13 La tendencia clave del software: Tecnología de objetos

Uno de los autores de este libro, HMD, recuerda la gran frustración que sentían las empresas de desarrollo de software, especialmente aquellas que desarrollaban proyectos a gran escala. Durante los veranos de sus años de estudiante, HMD tuvo el privilegio de trabajar en una empresa líder en la fabricación de computadoras como parte de los equipos de desarrollo de sistemas operativos con tiempo compartido y memoria virtual. Ésta fue una gran experiencia para un estudiante universitario. Sin embargo, en el verano de 1967, la realidad llegó cuando la empresa "decidió" producir de manera comercial el sistema en el que cientos de personas habían trabajado durante muchos años. Era difícil poner a punto el software. El software es un "asunto complejo".

Las mejoras a la tecnología de software comenzaron a aparecer con los beneficios de la denominada *programación estructurada* (y las disciplinas relacionadas como el *análisis y diseño de sistemas estructurados*) que se realizaba en la década de los setenta. Pero fue hasta que la tecnología de la programación orientada a objetos se hizo popular en la década de los noventa, que los desarrolladores de software sintieron que tenían las herramientas necesarias para realizar mayores adelantos en el proceso de desarrollo de software.

En realidad, la tecnología de objetos data de mediados de la década de los sesenta. El lenguaje de programación C++, desarrollado en AT&T por Bjarne Stroustrup a principios de la década de los ochenta, se basa en dos lenguajes: C, el cual se desarrolló inicialmente en AT&T a principios de la década de los sesenta para im-

plementar el sistema operativo UNIX, y Simula 67, un lenguaje de programación para simulación desarrollado en Europa y liberado en 1967. C++ absorbió las características de C y adicionó las capacidades de Simula para crear y manipular objetos. Ni C ni C++ se crearon originalmente para que se utilizaran fuera de los laboratorios de investigación de AT&T. Sin embargo, se desarrollaron con rapidez.

¿Qué son los objetos y por qué son tan especiales? En realidad, la tecnología de objetos es un esquema de compactación que permite crear unidades útiles de software. Éstas son grandes y altamente enfocadas a ámbitos de aplicación particulares. Existen objetos de fecha, de hora, de cheques, de facturas, de audio, de video, de archivo, de registro y de otros más. De hecho, casi cualquier sustantivo puede representarse razonablemente como un objeto.

Vivimos en un mundo de objetos. Sólo mire a su alrededor. Existen automóviles, aviones, gente, animales, edificios, semáforos, elevadores y otras cosas. Antes de la aparición de los lenguajes orientados a objetos, los lenguajes de programación (tales como FORTRAN, Pascal, Basic y C) se basaban en acciones (verbos), en lugar de cosas u objetos (sustantivos). Los programadores, que viven en un mundo de objetos, programan primordialmente mediante el uso de verbos. Este cambio de paradigma complicó la escritura de programas. Ahora, con la disponibilidad de los lenguajes orientados a objetos tales como Java y C++, los programadores siguen viviendo en un mundo orientado a objetos y pueden programar de una manera orientada a objetos. Éste es un proceso más natural de programación, y ha dado como resultado un mayor grado de productividad.

Un problema fundamental con la programación por procedimientos es que las unidades de programación no reflejan de manera sencilla y efectiva a las entidades del mundo real; así, estas unidades no son particularmente reutilizables. Con gran frecuencia, los programadores deben comenzar "de nuevo" cada nuevo proyecto y escribir código similar "desde cero". Esto significa un gasto de tiempo y de dinero, ya que la gente tiene que "reinventar la rueda" repetidamente. Mediante a tecnología de objetos, las entidades de software creadas (llamadas *clases*), si se diseñan apropiadamente, tienden a ser mucho más reutilizables en proyectos futuros. Con las bibliotecas de componentes reutilizables, tales como la *MFC (Microsoft Foundation Classes)* y las creadas por Rogue Wave y muchas otras empresas desarrolladoras de software, se puede reducir el esfuerzo requerido para implementar ciertas clases de sistemas (comparado con el esfuerzo que se hubiera requerido para reinventar estas capacidades en nuevos proyectos).

Algunas empresas indican que la reutilización de software no es, de hecho, el principal beneficio que obtienen de la programación orientada a objetos. Más bien, mencionan que la programación orientada a objetos tiende a producir software que es más comprensible, mejor organizado y fácil de mantener, modificar y corregir. Esto puede ser importante debido a que se estima que el 80% de los costos de software no están asociados con los esfuerzos originales para desarrollar software, sino que están asociados con la continua evolución y mantenimiento de ese software durante su vida útil.

Cualesquiera que sean los beneficios que se perciban de la programación orientada a objetos, es claro que ésta será la metodología clave de la programación en las siguientes décadas.

## 1.14 Conceptos básicos de un ambiente típico de programación en C

En general, los sistemas en C consisten en tres partes: un ambiente de desarrollo de programas, el lenguaje y la biblioteca estándar de C. La siguiente explicación define un ambiente típico de desarrollo en C como el que muestra la figura 1.1.

Los programas en C generalmente pasan a través de seis fases para ejecutarse (figura 1.1). Éstas son: *edición*, *preproceso*, *compilación*, *enlace*, *carga* y *ejecución*. Aunque *éste* es un texto genérico de C (escrito de manera independiente a los detalles de un sistema operativo en particular), en esta sección nos concentramos en un sistema de C basado en UNIX. [*Nota:* Los programas de este libro se ejecutarán con poca o sin modificación alguna en la mayoría de los sistemas comunes de C, los cuales incluyen sistemas basados en Windows de Microsoft.] Si usted no utiliza un sistema UNIX, consulte los manuales de su sistema, o pregunte a su profesor cómo llevar a cabo estas tareas en su ambiente.

La primera fase consiste en editar un archivo. Esto se lleva a cabo mediante un *programa de edición*. Dos editores ampliamente utilizados en sistemas UNIX son **`vi`** y **`emacs`**. Los paquetes de software para ambientes integrados de programación C/C++, tales como C++ Builder de Borland y Visual Studio de Microsoft, contienen editores que se encuentran integrados dentro del ambiente de programación. Asumimos que el lector sabe

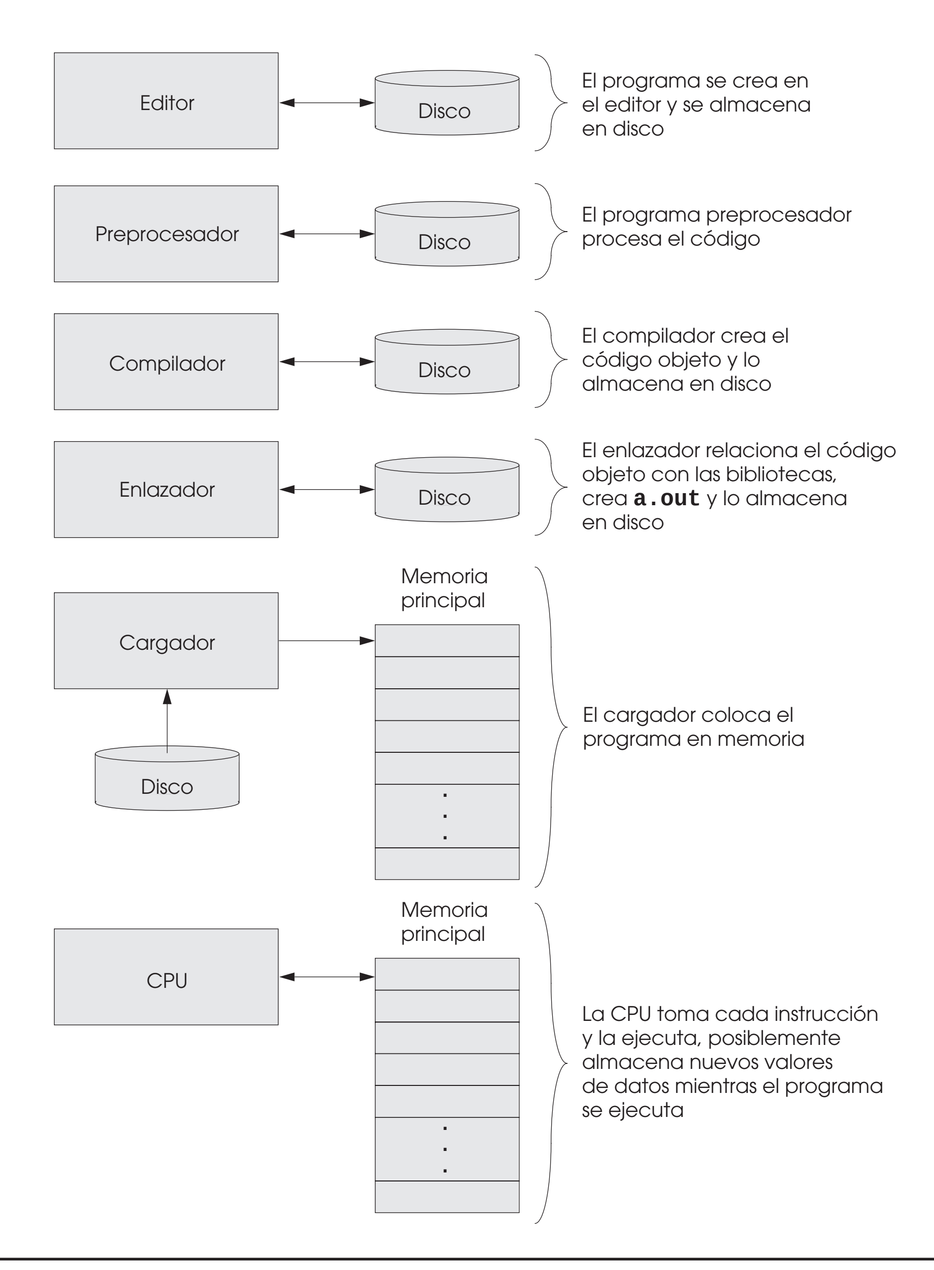



**Figura 1.1** Ambiente típico de desarrollo en C.

cómo editar un programa. El programador escribe un programa en C mediante un editor, hace correcciones si es necesario, y después almacena el programa en un dispositivo de almacenamiento secundario, como un disco. Los nombres de programas en C deben terminar con la extensión **`.c`**.

A continuación, el programador introduce el comando para *compilar* el programa. El compilador traduce el programa en C a código en lenguaje máquina (también conocido como *código objeto*). En un sistema de C, se ejecuta de manera automática un programa *preprocesador* antes de que comience la fase de traducción del compilador. El preprocesador de C obedece ciertos comandos especiales llamados *directivas del preprocesador*, las cuales indican que se deben realizar ciertas manipulaciones en el programa antes de la compilación. Por lo general, estas manipulaciones consisten en incluir otros archivos dentro del archivo para que sean compilados, y en realizar distintos reemplazos de texto. En los primeros capítulos explicaremos las directivas más comunes del preprocesador, y daremos una explicación detallada de las características del preprocesador en el capítulo 13.

El compilador invoca de manera automática al preprocesador, antes de que el programa sea convertido a lenguaje máquina.

La siguiente fase se denomina *enlace*. Por lo general, los programas en C contienen referencias a las funciones y datos definidos en alguna parte, tales como las bibliotecas estándar o las bibliotecas privadas de grupos de programadores que trabajan en un proyecto en particular. Por lo general, el código objeto producido por el compilador de C contiene "huecos", debido a estas partes faltantes. Un *enlazador* enlaza el código objeto con el código correspondiente a las funciones faltantes para producir una *imagen ejecutable* (sin piezas faltantes). En un típico sistema UNIX, el comando para compilar y enlazar un programa es **cc**. Para compilar y enlazar un programa llamado **bienvenido.c** teclee:

```
cc bienvenido.c
```

en el indicador de UNIX y presione la tecla *Entrar (Intro) (o de retorno)*. [*Nota:* Los comandos de UNIX son sensibles a mayúsculas y minúsculas, asegúrese de que teclea las *cs* como minúsculas y de que las letras del nombre de archivo sean mayúsculas o minúsculas, según sea el caso.] Si la compilación y el enlace del programa ocurren con éxito, se crea un archivo **a.out**. Ésta es una imagen ejecutable de nuestro programa **bienvenido.c**.

La siguiente fase se denomina *carga*. Antes de que el programa se pueda ejecutar, éste debe cargarse en memoria. Esto se lleva a cabo mediante el *cargador*, el cual toma la imagen ejecutable del disco y la transfiere a la memoria. También se cargan los componentes adicionales de las bibliotecas compartidas que soportan el programa.

Por último, la computadora, bajo el control de la CPU, *ejecuta* el programa, una instrucción a la vez. Para cargar y ejecutar el programa en un sistema UNIX, teclee **a.out** en el indicador de UNIX y presione *Entrar*.

Los programas no siempre funcionan al primer intento. Cada uno de los procedimientos puede fallar debido a distintos errores, los cuales explicaremos. Por ejemplo, un programa en ejecución podría intentar hacer una división entre cero (una operación ilegal en las computadoras, así como en la aritmética). Esto ocasionaría que la computadora desplegara un mensaje de error. El programador volvería entonces a la fase de edición, haría las correcciones necesarias y procedería con las fases restantes para verificar que correcciones funcionan adecuadamente.



**Error común de programación 1.1**

*Errores como la división entre cero ocurren durante la ejecución del programa, así que estos errores son denominados errores en tiempo de ejecución. En general, la división entre cero es un error fatal, es decir, un error que ocasiona la terminación inmediata del programa sin haber realizado de manera exitosa su trabajo. Los errores no fatales permiten al programa la ejecución completa, en su mayoría con resultados incorrectos.* [*Nota:* En algunos sistemas, la división entre cero no es un error fatal. Revise la documentación de su sistema.]

La mayoría de los programas en C introducen y/o arrojan datos. Ciertas funciones en C toman su entrada desde **stdin** (el *flujo estándar de entrada*) el cual es, por lo general, el teclado, pero el **stdin** puede conectarse a otro dispositivo. En su mayoría, los datos son arrojados hacia **stdout** (el *flujo estándar de salida*) el cual, por lo general es el monitor, pero el **stdout** puede conectarse a otro dispositivo. Cuando decimos que un programa imprime un resultado, normalmente nos referimos a que el resultado se despliega en el monitor. Los datos pueden ser arrojados hacia otros dispositivos tales como discos e impresoras de alta velocidad. Existe también un *flujo estándar de errores* denominado **stderr**. El flujo **stderr** (por lo general asociado con el monitor) se utiliza para desplegar los mensajes de error. Es común para los usuarios destinar los datos de salida normales, es decir, el **stdout**, hacia un dispositivo distinto al monitor y mantener el **stderr** asignado al monitor, de manera que el usuario pueda estar informado de los errores de manera inmediata.

## 1.15 Tendencias de hardware

La comunidad de programadores se desarrolla junto con el flujo continuo de avances dramáticos en el hardware, el software y las tecnologías de comunicación. En general, cada año la gente espera pagar más por la mayoría de los servicios y productos. Lo contrario ha sido el caso en los campos de las computadoras y las comunicaciones, especialmente con respecto a los costos de mantenimiento de estas tecnologías. Por muchas décadas, y sin expectativas de cambio alguno en un futuro próximo, los costos de hardware han disminuido de manera rápida, si no es que precipitada. Éste es un fenómeno de la tecnología. Cada uno o dos años, las capa-

cidades de las computadoras tienden a duplicarse mientras que los precios de las computadoras siguen cayendo. La disminución en picada de la relación costo/rendimiento de los sistemas de cómputo se debe a la creciente velocidad y capacidad de la memoria en la cual la computadora ejecuta sus programas, al aumento exponencial en la cantidad de memoria secundaria (tal como el almacenamiento en disco) en la que tienen que almacenar los programas y los datos durante largo tiempo, y al continuo incremento en la velocidad de proceso (la velocidad a la cual se ejecutan los programas en las computadoras, es decir, la velocidad a la que hacen su trabajo).

En las comunicaciones ha ocurrido el mismo crecimiento, y sus costos también han ido en picada, especialmente en años recientes con la enorme demanda por ancho de banda de comunicaciones, la cual atrae una enorme competencia. No conocemos otros campos en los que la tecnología se mueva tan rápidamente y los costos disminuyan de la misma forma. Cuando en las décadas de los sesenta y setenta hizo explosión el uso de las computadoras, se hablaba de las grandes mejoras en la productividad humana que las computadoras y las comunicaciones traerían consigo. Sin embargo, estas mejoras no se materializaron. Las empresas gastaron grandes sumas de dinero en computadoras, y con certeza las emplearon eficientemente, pero no vieron realizadas sus expectativas en cuanto a la productividad. Fue la invención de la tecnología de microprocesadores en chips y su amplia utilización a finales de la década de los setenta y en la de los ochenta, lo que sentó la base para las mejoras en la productividad actual.

## 1.16 Historia de Internet

A finales de la década de los sesenta, uno de los autores (HMD) de este libro era un estudiante egresado del MIT. Sus investigaciones dentro del proyecto Mac del MIT (ahora el laboratorio de ciencias de la computación, la casa del World Wide Web Consortium), eran patrocinadas por ARPA (Advanced Research Projects Agency of the Department of Defense). ARPA patrocinó una conferencia en la que algunas docenas de estudiantes del proyecto se reunieron en la universidad de Illinois, en Urbana-Champaign, para conocer y compartir sus ideas. Durante esta conferencia, ARPA difundió el anteproyecto de conectar en red a las principales computadoras de una docena de universidades e institutos de investigación patrocinados por ARPA. Éstas se conectarían mediante líneas de comunicación que operaban, en ese entonces, a la increíble velocidad de 56 KB (es decir, 56,000 bits por segundo), esto en una época en la que la mayoría de la gente (de los pocos que podían estarlo) se conectaba mediante las líneas telefónicas a las computadoras a un rango de velocidad de 110 bits por segundo. HMD recuerda lúcidamente la emoción en aquella conferencia. Investigadores de Harvard hablaron acerca de comunicar la Univac 1108, "una supercomputadora" de la universidad de Utah, con todo el país, para manejar los cálculos relacionados con sus investigaciones acerca de gráficos por computadora. Se comentaron muchas otras posibilidades intrigantes. La investigación académica estaba a punto de dar un paso gigantesco hacia delante. Poco después de ésta conferencia, ARPA procedió con la implantación de lo que pronto se convirtió en *ARPAnet*, el abuelo de la *Internet* actual.

Las cosas resultaron diferentes a lo que se había planeado originalmente. En lugar de que el principal beneficio fuera el que los investigadores pudieran compartir sus computadoras, se hizo evidente que el principal beneficio de *ARPAnet* iba a ser el permitir que los investigadores se comunicaran de una manera rápida y fácil entre ellos, por medio de lo que se llamó *correo electrónico (e-mail)*. Esto es verdad incluso en el Internet actual, en donde el correo electrónico facilita la comunicación de todo tipo de personas alrededor del mundo.

Una de las principales metas de ARPA, con respecto a la red, era permitir que múltiples usuarios enviaran y recibieran información al mismo tiempo y sobre las mismas rutas de comunicación (tal como una línea telefónica). La red operaba mediante una técnica denominada *intercambio de paquetes*, en la cual, un dato digital se enviaba en pequeños *paquetes*. Dichos paquetes contenían datos, información de la dirección, información para el control de errores y la información de la secuencia. La información sobre la dirección se utilizaba para establecer la ruta de los paquetes hacia su destino. La información de la secuencia se utilizaba para ayudar a acomodar los paquetes en su orden original (los cuales, debido a los complejos mecanismos de ruteo, en realidad pueden llegar en desorden). Los paquetes de muchas personas se mezclaban en las mismas líneas de comunicación. La técnica de intercambio de paquetes redujo de manera importante los costos de transmisión, comparados con los costos de las líneas de comunicación dedicadas.

La red se diseñó para operar sin un control central. Esto significaba que si una porción de la red fallaba, las porciones restantes podrían ser capaces de enviar paquetes, de los remitentes a los destinatarios, a través de rutas alternas.

Los protocolos para la comunicación a través de ARPA*net* se hicieron conocidos como *TCP (Transmission Control Protocol)*. TCP garantizaba que los mensajes se enrutaran apropiadamente del remitente al destinatario, y que los mensajes llegaran intactos.

En paralelo con la primera evolución de Internet, las empresas de todo el mundo estaban instalando sus propias redes de comunicación, tanto intraempresariales (dentro de la empresa), como interempresariales (entre las empresas). En ese entonces apareció una gran cantidad de hardware y software para redes. Uno de los desafíos era lograr la intercomunicación. ARPA lo logró mediante el desarrolló de *IP (Internet Protocol)*, y con ello creó la verdadera "red de redes"; la arquitectura actual de Internet. A la combinación de ambos protocolos se le denomina *TCP/IP*.

En un principio, el uso de Internet estaba limitado a las universidades y a los institutos de investigación; después, la milicia se convirtió en un usuario importante. En algún momento, el gobierno permitió el acceso a Internet con fines comerciales. De entrada, hubo recelo por parte de las comunidades militares y de investigación, pensaban que el tiempo de respuesta se haría deficiente, conforme "la red" se saturara de usuarios.

De hecho, ha ocurrido lo contrario. La gente de negocios rápidamente se dio cuenta de que si utilizaban efectivamente la Internet, podrían afinar sus operaciones y ofrecer nuevos y mejores servicios a sus clientes. Como resultado, los ejecutivos de negocios gastaron grandes cantidades de dinero para desarrollar y mejorar Internet. Esto generó una feroz competencia entre los proveedores de dispositivos de comunicación, de hardware y software para cubrir la demanda. El resultado es que el *ancho de banda* (es decir, la capacidad de transmisión de información de las líneas de comunicación) sobre Internet ha crecido enormemente y los costos han ido en picada. En la actualidad, los países alrededor del mundo saben que Internet es crucial para su prosperidad económica y su competitividad.

## 1.17 Historia de la World Wide Web

La *World Wide Web* permite a los usuarios de computadoras, localizar y ver documentos basados en multimedia (es decir, documentos con texto, gráficos, animación, audio y/o video) de casi cualquier tema. Aunque Internet se desarrolló hace más de tres décadas, la introducción de *World Wide Web* es un suceso relativamente reciente. En 1990, *Tim Berners-Lee,* miembro de la CERN (European Organization for Nuclear Research) desarrolló la World Wide Web y los distintos protocolos de comunicación que forman su esqueleto.

Tanto Internet como la World Wide Web estarán en la lista de las creaciones más importantes de la humanidad. En el pasado, la mayoría de las aplicaciones de cómputo se ejecutaban sobre computadoras "independientes", es decir, computadoras que no estaban conectadas entre sí. Las aplicaciones actuales pueden ser escritas para comunicar a cientos de miles de computadoras alrededor del mundo. Internet combina las tecnologías de comunicación y computación. Hace más fácil nuestro trabajo. Hace que la información esté disponible de manera instantánea y conveniente a nivel mundial. Hace posible que los individuos y los pequeños negocios puedan exponerse a nivel mundial. Está modificando la naturaleza de la forma en que se llevan a cabo los negocios. La gente puede buscar los mejores precios y virtualmente cualquier producto o servicio. Las comunidades con intereses especiales pueden mantenerse en contacto entre sí. Los investigadores pueden dar aviso de manera instantánea de los últimos avances a nivel mundial.

## 1.18 Notas generales acerca de C y de este libro

Algunas veces, los programadores experimentados de C se sienten orgullosos por ser capaces de crear aplicaciones raras, retorcidas e intrincadas del lenguaje. Ésta es una mala práctica de programación. Hace que los programas sean más difíciles de leer, con mayor probabilidad de comportarse de manera extraña, más difíciles de leer y depurar, y más difíciles de adaptar a modificaciones necesarias. Este libro se orienta hacia los programadores principiantes, por ello motivamos la *claridad*. La siguiente es nuestra primera "buena práctica de programación".



**Buena práctica de programación 1.1**

*Escriba sus programas en C de manera clara, directa y simple. A esto se le llama algunas veces KIS ("keep it simple", manténgalo simple). No "estire" el lenguaje, intentando emplearlo de manera extraña.*

Probablemente ha escuchado que C es un lenguaje portable, y que los programas escritos en C pueden ejecutarse en muchas computadoras diferentes. La *portabilidad* es una *meta escurridiza*. El documento C están-

dar de ANSI contiene una larga lista de temas acerca de la portabilidad, y se han escrito libros completos que la explican.



**Tip de portabilidad 1.3**

*Aunque es posible escribir programas portables, existen muchos problemas entre los diferentes compiladores de C, y las computadoras pueden hacer que la portabilidad sea difícil de conseguir. Escribir programas en C no garantiza la portabilidad. A menudo, el programador tendrá que enfrentarse directamente con las variaciones entre los compiladores y las computadoras.*

Nosotros hicimos una revisión cuidadosa del documento para el estándar de C, y comparamos nuestra presentación contra este documento para que fuera completa y acertada. Sin embargo, C es un lenguaje rico, y existen algunas sutilezas en el lenguaje y algunos temas avanzados que no cubrimos. Si usted requiere detalles técnicos adicionales sobre C, le sugerimos que lea el documento de C estándar o el libro de Kernighan y Ritchie.

Nosotros limitamos nuestra explicación al C de ANSI/ISO. Muchas de las características de esta versión de C no son compatibles con implementaciones antiguas de C, de manera que algunos de los programas en este texto podrán no funcionar en antiguos compiladores de C.



**Observación de ingeniería de software 1.1**

*Lea los manuales de la versión de C que utiliza. Consulte estos manuales con frecuencia para percatarse de la rica colección de características de C y para que las utilice de manera correcta.*



**Observación de ingeniería de software 1.2**

*Su computadora y su compilador son buenos maestros. Si no está seguro de cómo funciona alguna característica de C, escriba un programa sencillo con dicha característica, compile y ejecute el programa para que vea qué sucede.*

## RESUMEN

- El software (es decir, las instrucciones que usted escribe para indicar a la computadora que realice acciones y tome decisiones) controla a las computadoras (a menudo conocidas como hardware).
- El C de ANSI es la versión del lenguaje de programación que se estandarizó en 1989, tanto para los Estados Unidos a través del American National Standards Institute (ANSI) y alrededor del mundo a través del International Standards Organization (ISO).
- Las computadoras que antes ocupaban grandes habitaciones y costaban millones de dólares años atrás, ahora se pueden introducir en la superficie de chips de silicio más pequeños que una uña y su costo es quizá de unos cuantos dólares cada una.
- Cientos de millones de computadoras de uso general se emplean a lo largo del mundo para ayudar a la gente en las empresas, la industria, el gobierno y sus vidas personales. Dicho número podría duplicarse fácilmente en unos cuantos años.
- Una computadora es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones de veces más rápido que los humanos.
- Las computadoras procesan los datos bajo el control de los programas de cómputo.
- A los distintos dispositivos (tales como las unidades de teclado, pantalla, discos, memoria y proceso) que componen un sistema de cómputo se les conoce como hardware.
- A los programas de cómputo que se ejecutan en una computadora se les conoce como software.
- La unidad de entrada es la sección “receptora” de la computadora. En la actualidad, la mayor parte de la información se introduce a las computadoras mediante teclados parecidos a máquinas de escribir.
- La unidad de salida es la sección de “envío” de la computadora. En la actualidad, la mayor parte de la información sale de las computadoras desplegándola en pantalla o imprimiéndola en papel.
- La unidad de memoria es la sección de “almacenaje” de la computadora, y a menudo se le denomina memoria o memoria principal.
- La unidad aritmética y lógica (ALU) realiza los cálculos y toma las decisiones.
- La unidad central de procesamiento (CPU) es la administradora de la computadora y es la responsable de supervisar la operación de las otras secciones.
- Por lo general, los programas y los datos que no se utilizan de manera activa por las otras unidades se colocan en dispositivos de memoria secundaria (tales como discos) hasta que nuevamente son requeridos.
- Los sistemas operativos son sistemas de software que facilitan el uso de las computadoras y la obtención de un mejor rendimiento.
- Los sistemas operativos con multiprogramación permiten la operación “simultánea” de muchas tareas en la computadora, la computadora comparte sus recursos entre las diferentes tareas.

- El tiempo compartido es un caso especial de la multiprogramación en la cual, los usuarios acceden a la computadora a través de terminales. Los usuarios parecen ejecutar sus tareas de manera simultánea.
- Mediante la computación distribuida, el cómputo de una empresa se distribuye mediante la red a los sitios en donde se realiza el trabajo de la empresa.
- Los servidores almacenan programas y datos que se pueden compartir con las computadoras clientes, distribuidas a lo largo de la red; de ahí el término computación cliente-servidor.
- Cualquier computadora sólo puede comprender de manera directa su propio lenguaje máquina. Por lo general, los lenguajes máquina constan de cadenas de números (cadenas de unos y ceros) que indican a la computadora que realice las operaciones más elementales, una a la vez. Los lenguajes máquina son dependientes de la máquina.
- Las abreviaturas del inglés forman la base de los lenguajes ensambladores. Los ensambladores traducen los programas en lenguaje ensamblador a lenguaje máquina.
- Los compiladores traducen programas en lenguajes de alto nivel a lenguaje máquina. Los lenguajes de alto nivel contienen palabras en inglés y notaciones matemáticas convencionales.
- Los programas intérpretes ejecutan de manera directa programas de alto nivel, sin la necesidad de compilar dichos programas a lenguaje máquina.
- Aunque los programas compilados se ejecutan más rápidamente que los programas intérpretes, los intérpretes son populares en ambientes de desarrollo de programas, en los cuales los programas se recompilan con frecuencia mientras se adicionan nuevas características y se corrigen errores. Una vez que se desarrolla un programa, se puede producir una versión compilada que se ejecuta de manera más eficiente.
- FORTRAN (FORmula TRANslator) se utiliza para aplicaciones matemáticas. COBOL (COmmon Business Oriented Language) se utiliza primordialmente para aplicaciones comerciales que requieren una manipulación precisa y eficiente de grandes cantidades de datos.
- La programación estructurada es un método disciplinado para escribir programas más claros, más fáciles de probar, depurar y modificar, que los programas no estructurados.
- Pascal fue diseñado para enseñar programación estructurada.
- Ada se desarrolló bajo el patrocinio del departamento de defensa de Estados Unidos (DoD), utilizando Pascal como base. A Lady Lovelace se le da el crédito de haber escrito el primer programa a principios de 1800 (para la Máquina Analítica de cómputo diseñada por Charles Babbage).
- Las multitareas permite a los programadores especificar actividades en paralelo.
- A C se le conoce como el lenguaje de desarrollo del sistema operativo UNIX.
- Es posible escribir programas de C que son portables a la mayoría de las computadoras.
- Existen dos claves para aprender a programar en C. La primera es aprender el propio lenguaje C, y la segunda es aprender cómo utilizar las funciones de la biblioteca estándar de C.
- C++ es un conjunto ampliado de C, desarrollado por Bjarne Stroustrup en los laboratorios Bell. C++ proporciona las capacidades para la programación orientada a objetos.
- Los objetos son esencialmente componentes reutilizables de software que modelan elementos del mundo real.
- Utilizar un método de diseño e implementación modular y orientado a objetos puede hacer que los grupos de desarrollo de software sean más productivos que con técnicas convencionales de programación.
- Java se utiliza para crear páginas Web con contenido dinámico e interactivo, desarrollar aplicaciones empresariales a gran escala, aumentar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en nuestros exploradores Web), proporcionar aplicaciones para los dispositivos del consumidor (tales como teléfonos celulares, localizadores y asistentes personales digitales).
- El lenguaje de programación BASIC (Beginner´s All-Purpose Symbolic Instruction Code) fue desarrollado a mediados de la década de los sesenta por los profesores del Darmouth Collage John Kemeny y Thomas Kurtz, como un lenguaje para escribir programas sencillos. El propósito principal de BASIC era familiarizar a los principiantes con las técnicas de programación.
- Visual Basic .NET, Visual C++ .NET y C# fueron diseñados para la nueva plataforma de programación de Microsoft, .NET. Los tres lenguajes utilizan la poderosa biblioteca de componentes reutilizables de .NET llamada Framework Class Library (FCL).
- El lenguaje de programación C# fue diseñado por Microsoft de manera específica para su plataforma .NET, como un lenguaje que permitiera a los programadores migrar fácilmente a .NET.
- Comparada con Java, la plataforma .NET permite a las aplicaciones basadas en Web ser distribuidas a muchos dispositivos (incluso teléfonos celulares) y computadoras de escritorio.
- C++, Java y C# tienen sus raíces en el lenguaje de programación C.
- La tecnología de objetos es un esquema de empaquetamiento que nos ayuda a crear unidades de software útiles. Éstas son grandes y muy enfocadas a campos de aplicación en particular.

- Un problema clave con la programación por procedimientos es que las unidades de programación no reflejan con facilidad entidades del mundo real, de manera que dichas unidades no son particularmente reutilizables. No es poco común para los programadores "comenzar de cero" cada proyecto y tener que escribir software similar "desde cero".
- Mediante la tecnología de objetos, las entidades de software creadas (llamadas *clases*), si se diseñan de manera correcta, tienden a ser más reutilizables para proyectos futuros. Utilizar bibliotecas de componentes reutilizables puede reducir en gran medida el esfuerzo requerido para implementar ciertos tipos de sistemas (comparado con el esfuerzo que requeriría reinventar estas capacidades en un nuevo proyecto).
- La programación orientada a objetos tiende a producir software más comprensible, mejor organizado y más fácil de mantener, modificar y depurar. Esto puede ser importante debido a que se estima que aproximadamente el 80% de los costos de software están asociados con la continua evaluación y mantenimiento de dicho software a través de su vida útil.
- Todos los sistemas en C constan de tres partes: el ambiente, el lenguaje y las bibliotecas estándar. Las funciones de la biblioteca no son parte del propio lenguaje C; estas funciones realizan operaciones tales como entrada/salida y cálculos matemáticos.
- Por lo general, los programas en C pasan a través de seis fases para su ejecución: edición, preproceso, compilación, enlace, carga y ejecución.
- El programador escribe un programa mediante un editor y hace las correcciones necesarias. Por lo general, los nombres de archivos en C terminan con la extensión **`.c`**.
- Un compilador traduce un programa en C a lenguaje máquina (o código objeto).
- El preprocesador de C obedece las directivas del preprocesador, las cuales indican la inclusión de otros archivos dentro del archivo a compilar y que los símbolos especiales se reemplazarán por texto del programa.
- Un enlazador enlaza el código objeto con el código de las funciones faltantes para producir una imagen ejecutable (sin piezas faltantes). En un sistema típico basado en UNIX, el comando para compilar y enlazar un programa en C es **`cc`**. Si el programa se compila y se enlaza de manera correcta, se produce un archivo llamado **`a.out`**. Ésta es la imagen ejecutable del programa.
- Un cargador toma una imagen ejecutable desde el disco y la transfiere a la memoria.
- Errores como la división entre cero ocurren durante la ejecución del programa, por tal motivo se les conoce como errores en tiempo de ejecución.
- Por lo general, a la división entre cero se le considera como error fatal, es decir, un error que provoca la terminación inmediata del programa sin haber terminado satisfactoriamente su trabajo. Los errores no fatales permiten a los programas ejecutarse por completo, a menudo con la producción de resultados incorrectos.
- Una computadora, bajo el control de su CPU, ejecuta un programa instrucción por instrucción.
- Ciertas funciones en C (como **`scanf`**) toman su entrada desde **`stdin`** (el flujo estándar de entrada), el cual está, por lo general, asignado al teclado. Los datos son arrojados hacia **`stdout`** (el flujo estándar de salida) el cual está, por lo general, asignado a la pantalla de la computadora.
- También existe un flujo estándar de errores denominado **`stderr`**. El flujo **`stderr`** (por lo general asignado a la pantalla) se utiliza para desplegar mensajes de error.
- Existen muchas variaciones entre las diferentes implementaciones de C y las diferentes computadoras, lo que hace de la portabilidad una meta escurridiza.

## *TERMINOLOGÍA*

Ada
ALU
ambiente
BASIC
biblioteca de clases
biblioteca estándar de C
bibliotecas estándar
C
C#
C++
cargador
claridad
cliente
COBOL
código objeto
compilador
componentes reutilizables de software
computación cliente/servidor
computación distribuida
computadora
computadora personal
CPU
dato
dependiente de la máquina
depuración
dispositivo de entrada
dispositivo de salida
editor
ejecutar un programa
enlazador
ensamblador
entrada/salida (E/S)
error en tiempo de ejecución
error fatal
error no fatal
estándar C de ANSI/ISO
extensión **`.c`**
flujo de entrada
flujo de salida
flujo estándar de entrada (**`stdin`**)
flujo estándar de errores (**`stderr`**)
flujo estándar de salida (**`stdout`**)
FORTRAN
Framework Class Library (FCL)

función
función de biblioteca
hardware
imagen ejecutable
independiente de la máquina
Internet
Java
KIS (“keep it simple”)
Lady Ada Lovelace
lenguaje de alto nivel
lenguaje de programación
lenguaje ensamblador
lenguaje máquina
lenguaje natural de una computadora
Linux
mejoramiento paso a paso
memoria
memoria principal
método de construcción por bloques
multiprocesador
multiprogramación
multitareas
.NET
objeto
OS X de Mac
pantalla
Pascal
plataforma de hardware
portabilidad
preprocesador
preprocesador de C
procesamiento por lotes
programa
programa almacenado
programa de computadora
programa intérprete
programa traductor
programación estructurada
programación orientada a objetos (POO)
programador de computadoras
redes de computadoras
rendimiento
reutilización de software
servidor de archivos
sistema operativo
software
subprocesamiento múltiple
supercomputadora
tarea
TCP/IP
terminal
tiempo compartido
unidad central de procesamiento (CPU)
unidad de entrada
unidad de memoria
unidad de memoria secundaria
unidad de salida
unidad aritmética y lógica (ALU)
unidades lógicas
UNIX
Visual Basic .NET
Visual C++
Visual C++.NET
Windows
World Wide Web

## ERROR COMÚN DE PROGRAMACIÓN

**1.1** Errores como la división entre cero ocurren durante la ejecución del programa, así que estos errores son denominados errores en tiempo de ejecución. Generalmente, la división entre cero es un error fatal, es decir, un error que ocasiona la terminación inmediata del programa sin haber realizado de manera exitosa su trabajo. Los errores no fatales permiten al programa la ejecución completa, en su mayoría con resultados incorrectos. (*Nota:* En algunos sistemas, la división entre cero no es un error fatal. Revise la documentación de su sistema.)

## BUENA PRÁCTICA DE PROGRAMACIÓN

**1.1** Escriba sus programas en C de manera clara, directa y simple. A esto se le llama algunas veces KIS (“keep it simple”, manténgalo simple). No “estire” el lenguaje, intentando emplearlo de manera extraña.

## TIP DE RENDIMIENTO

**1.1** Utilizar funciones de la biblioteca estándar de ANSI, en lugar de escribir sus propias funciones similares, puede mejorar el rendimiento del programa debido a que estas funciones están escritas cuidadosamente para una ejecución eficiente.

## TIPS DE PORTABILIDAD

**1.1** Debido a que C es un lenguaje ampliamente disponible, independiente de la plataforma, y estandarizado, las aplicaciones escritas en C a menudo pueden ejecutarse sobre un amplio rango de sistemas de cómputo con muy pocas o ninguna modificación.

**1.2** Utilizar funciones de la biblioteca estándar de ANSI, en lugar de escribir sus propias funciones similares, puede mejorar la portabilidad debido a que estas funciones se utilizan virtualmente en cualquier implementación del C de ANSI.

**1.3** Aunque es posible escribir programas portables, existen muchos problemas entre los diferentes compiladores de C, y las computadoras pueden hacer que la portabilidad sea difícil de conseguir. Escribir programas en C no garantiza la portabilidad. A menudo, el programador tendrá que enfrentarse directamente con las variaciones entre los compiladores y las computadoras.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**1.1** Lea los manuales para la versión de C que utiliza. Consulte estos manuales con frecuencia para percatarse de la rica colección de características de C y para que las utilice de manera correcta.

**1.2** Su computadora y su compilador son buenos maestros. Si no está seguro de cómo funciona alguna característica de C, escriba un programa sencillo con dicha característica, compile y ejecute el programa para que vea qué sucede.

## EJERCICIOS DE AUTOEVALUACIÓN

**1.1** Complete los espacios en blanco:

a) La empresa que provocó el fenómeno mundial de la computación personal fue ________.
b) La computadora que dio legitimidad a la computación personal en las empresas y en la industria fue la ________ .
c) Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamados ________ .
d) Las seis unidades lógicas clave de la computadora son: ________, ________, ________, ________, ________ y ________.
e) El ________ es un caso especial de la multiprogramación, en la que los usuarios acceden a la computadora a través de dispositivos llamados terminales.
f) Los tres tipos de lenguajes explicados en este capítulo son ________, ________ y ________.
g) A los programas que traducen programas escritos en un lenguaje de alto nivel a lenguaje máquina se les llama ________.
h) A C se le conoce ampliamente como el lenguaje de desarrollo del sistema operativo ________.
i) Este libro presenta la versión de C conocida como ________ C que recientemente fue estandarizada a través de la American National Standards Institute.
j) El lenguaje ________ fue desarrollado por Wirth para la enseñanza de la programación estructurada.
k) El departamento de defensa de los Estados Unidos desarrolló el lenguaje Ada con una capacidad llamada ________, la cual permite a los programadores especificar la realización de varias tareas en paralelo.

**1.2** Complete los espacios en blanco de cada una de las siguientes frases acerca del ambiente C.

a) Por lo general, los programas en C se introducen a la computadora mediante el uso de un programa ________.
b) En un sistema C, un programa ________ se ejecuta de manera automática antes de que comience la fase de traducción.
c) Los dos tipos más comunes de directivas de preprocesador son ________ y ________.
d) El programa ________ combina la salida del compilador con varias bibliotecas de funciones para producir una imagen ejecutable.
e) El programa ________ transfiere la imagen ejecutable desde el disco a la memoria.
f) Para cargar y ejecutar el programa más recientemente compilado en un sistema UNIX, teclee ________.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**1.1** a) Apple. b) Computadora personal de IBM. c) Programas. d) Unidad de entrada, unidad de salida, unidad de memoria, unidad aritmética y lógica (ALU), unidad central de procesamiento (CPU), unidad de almacenamiento secundario. e) Tiempo compartido. f) Lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. g) Compiladores. h) UNIX. i) ANSI. j) Pascal. k) Multitareas.

**1.2** a) Editor. b) Preprocesador. c) Incluir otros archivos dentro del archivo a compilar, reemplazar símbolos especiales con texto del programa. d) Enlazador. f) `a.out`.

## EJERCICIOS

**1.3** Clasifique cada uno de los elementos siguientes como hardware o software:

a) CPU.
b) Compilador de C.
c) ALU.
d) Preprocesador de C.
e) Unidad de entrada.
f) Programa procesador de texto.

**1.4** ¿Por qué querría usted escribir un programa en un lenguaje independiente de la máquina, en lugar de hacerlo en un lenguaje dependiente de la máquina? ¿Por qué sería más apropiado escribir cierto tipo de programas en un lenguaje dependiente de la máquina?

**1.5** Los programas traductores tales como ensambladores y compiladores convierten los programas de un lenguaje (llamado código *fuente*) a otro lenguaje (llamado código *objeto*). Determine cuál de las siguientes frases es verdadera y cual es falsa:

a) Un compilador traduce programas en un lenguaje de alto nivel a código objeto.
b) Un ensamblador traduce programas en código fuente a programas en lenguaje máquina.
c) Un compilador convierte programas en código fuente a programas en código objeto.
d) Por lo general, los lenguajes de alto nivel son dependientes de la máquina.
e) Un programa en lenguaje máquina requiere traducción antes de poderlo ejecutar en una computadora.

**1.6** Complete los espacios en blanco:

a) Por lo general, a los dispositivos desde los cuales los usuarios acceden a sistemas de cómputo de tiempo compartido se les llama ____________.
b) A un programa de cómputo que convierte programas en lenguaje ensamblador a programas en lenguaje máquina se le llama ____________.
c) A la unidad lógica de la computadora que recibe información desde fuera para que la utilice se le llama ____________.
d) Al proceso de instruir a la computadora para resolver un problema específico se le llama ____________.
e) ¿Qué tipo de lenguaje de cómputo utiliza abreviaturas parecidas al inglés para instrucciones en lenguaje máquina? ____________.
f) ¿Qué unidad lógica de la computadora envía la información procesada por la computadora hacia varios dispositivos, de manera que la información se pueda utilizar fuera de ella? ____________.
g) El nombre general para un programa que convierte programas escritos en cierto lenguaje de computadora a lenguaje máquina es ____________.
h) ¿Cuál unidad lógica de la computadora retiene la información? ____________.
i) ¿Cuál unidad lógica de la computadora realiza los cálculos? ____________.
j) ¿Cuál unidad lógica de la computadora toma decisiones lógicas? ____________.
k) La abreviatura común, utilizada para la unidad de control de la computadora es ____________.
l) El nivel más conveniente de un lenguaje de computadora para que un programador escriba programas rápida y fácilmente es ____________.
m) Al único lenguaje que una computadora puede comprender directamente se le llama ____________.
n) ¿Cuál unidad lógica de la computadora coordina las actividades de las otras unidades lógicas? ____________.

**1.7** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique su respuesta.

a) Por lo general, los lenguajes de máquina son dependientes de la máquina.
b) El tiempo compartido realmente permite la ejecución simultánea de las tareas de varios usuarios en una misma computadora.
c) Como a otros lenguajes de alto nivel, a C generalmente se le considera independiente de la máquina.

**1.8** Explique el significado de cada uno de los siguientes nombres:

a) **`stdin`**
b) **`stdout`**
c) **`stderr`**

**1.9** ¿Por qué en la actualidad existe tanta atención centrada a la programación orientada a objetos en lo general y en C++ en lo particular?

**1.10** ¿Cuál lenguaje de programación describe mejor cada una de las siguientes frases?

a) Desarrollado por IBM para aplicaciones científicas y de ingeniería.
b) Desarrollado específicamente para aplicaciones de negocios.
c) Desarrollado para la enseñanza de la programación estructurada.
d) Su nombre tiene origen en el primer programador del mundo.
e) Desarrollado para introducir a los novatos en las técnicas de programación.
f) Desarrollado específicamente para ayudar a los programadores a migrar a .NET.
g) Conocido como el lenguaje de desarrollo de UNIX.
h) Creado principalmente añadiendo a C capacidades para programación orientada a objetos.
i) Inicialmente tuvo éxito debido a su habilidad para crear páginas Web con contenido dinámico.



# Introducción a la programación en C

## Objetivos

- Escribir programas sencillos en C.
- Utilizar instrucciones sencillas de entrada y salida.
- Familiarizarse con los tipos de datos fundamentales.
- Comprender conceptos sobre la memoria de las computadoras.
- Utilizar los operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Escribir instrucciones condicionales sencillas.

*¿Qué hay en un nombre? Eso que llamamos rosa*
*Para cualquier otro nombre olería muy dulce.*
William Shakespeare *Romeo y Julieta*

*Yo sólo tomé el curso normal… las diferentes ramas de*
*la aritmética —ambición, distracción, afeamiento y escarnio.*
Lewis Carroll

*Los precedentes deliberadamente establecidos por hombres sabios*
*merecen gran valor.*
Henry Clay



## Plan general



## 2.1 Introducción

El lenguaje C facilita un método estructurado y disciplinado para el diseño de programas. En este capítulo introducimos la programación en C y presentamos varios ejemplos que ilustran muchas características importantes de C. Analizamos cuidadosamente cada ejemplo, línea por línea. En los capítulos 3 y 4 presentamos una introducción a la *programación estructurada* en C. Después utilizamos dicho método estructurado en el resto del libro.

## 2.2 Un programa sencillo en C: Impresión de una línea de texto

C utiliza una notación que puede parecer extraña para quien no es programador. Comencemos considerando un programa sencillo en C. Nuestro primer ejemplo imprime una línea de texto. El programa y su resultado en pantalla aparecen en la figura 2.1.

Aun cuando este programa es sencillo, ilustra muchas características importantes del lenguaje C. Ahora consideremos con detalle cada línea del programa. Las líneas 1 y 2:

```
/* Figura 2.1: fig02_01.c
   Un primer programa en C */
```

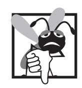
comienzan con **/*** y terminan con ***/**, lo que indica que estas dos líneas son un *comentario*. Los programadores insertan comentarios para *documentar* los programas y para mejorar su legibilidad. Los comentarios no provocan que la computadora realice acción alguna durante la ejecución del programa. El compilador de C ignora

```
1  /* Figura 2.1: fig02_01.c
2     Un primer programa en C */
3  #include <stdio.h>
4
5  /* la función main inicia la ejecución del programa */
6  int main( void )
7  {
8     printf( “Bienvenido a C!\n” );
9
10    return 0; /* indica que el programa terminó con éxito */
11
12 } /* fin de la función main */
```

```
Bienvenido a C!
```

**Figura 2.1** Programa de impresión de texto.

los comentarios y no genera código objeto en lenguaje máquina. El comentario anterior sólo describe el número de la figura, el nombre del archivo y el propósito del programa. Los comentarios también ayudan a otras personas a leer y entender un programa, pero demasiados comentarios pueden ocasionar que un programa sea difícil de leer.



**Error común de programación 2.1**

*Olvidar finalizar un comentario con* `*/`.



**Error común de programación 2.2**

*Comenzar un comentario con los caracteres* `*/`*, o finalizarlo con* `/*`.

La línea 3

```
#include <stdio.h>
```

es una directiva del *preprocesador* de C. Las líneas que comienzan con **#** son procesadas por el preprocesador antes de que el programa se compile. Esta línea en particular indica al preprocesador que incluya en el programa el contenido del *encabezado estándar de entrada/salida* (**stdio.h**). Este encabezado contiene información que el compilador utiliza cuando compila las llamadas a las funciones de la biblioteca estándar de entrada/salida, como **printf**. En el capítulo 5 explicaremos con más detalle el contenido de los encabezados.

La línea 6

```
int main( )
```

forma parte de todos los programas en C. Los paréntesis que aparecen después de **main** indican que **main** es un bloque de construcción de programas llamado *función*. Los programas en C contienen una o más funciones, una de las cuales debe ser **main**. Todo programa en C comienza su ejecución en la función **main**.



**Buena práctica de programación 2.1**

*Toda función debe ser precedida por un comentario que describa el propósito de la función.*

La *llave izquierda*, **{**, (línea 7), debe iniciar el *cuerpo* de cada función. Una *llave derecha* correspondiente (línea 12), debe finalizar cada función. Este par de *llaves* y la parte del programa entre ellas se conocen como *bloque*. El bloque es una unidad importante del programa en C.

La línea 8

```
printf( “Bienvenido a C!\n” );
```

indica a la computadora que realice una *acción*, es decir, que imprima en la pantalla la *cadena* de caracteres contenida entre las comillas. En algunas ocasiones a una cadena se le llama *cadena de caracteres, mensaje* o *literal*. La línea completa [que incluye **printf**, su *argumento* entre paréntesis, y el *punto y coma* (**;**)] se conoce como *instrucción*. Toda instrucción debe finalizar con un punto y coma (también conocido como *terminador de la instrucción*). Cuando la instrucción **printf** anterior se ejecuta, ésta imprime en la pantalla el mensaje **Bienvenido a C!** En general, los caracteres se imprimen exactamente como aparecen entre las comillas de la instrucción **printf**. Observe que los caracteres **\n** no aparecieron en pantalla. La diagonal invertida (****) se conoce como *carácter de escape*. Éste indica que se espera que **printf** haga algo fuera de lo ordinario. Cuando una diagonal invertida se encuentra dentro de una cadena, el compilador ve el siguiente carácter y lo combina con la diagonal invertida para formar una *secuencia de escape*. La secuencia de escape **\n** significa *nueva línea*. Cuando una nueva línea aparece en la salida de la cadena por medio de **printf**, esta nueva línea ocasiona que el cursor se posicione al comienzo de la siguiente línea de la pantalla. En la figura 2.2 aparecen algunas secuencias de escape comunes.

Las dos últimas secuencias de escape de la figura 2.2 pueden parecer extrañas. Debido a que la diagonal invertida tiene un significado especial en una cadena, es decir, que el compilador la reconoce como un carácter de escape, nosotros utilizamos dos diagonales invertidas para colocar una sola diagonal invertida en una cadena. Imprimir comillas también representa un problema, ya que dichas comillas marcan el límite de una cadena; de

| Secuencia de escape | Descripción |
|---|---|
| `\n` | Nueva línea. Coloca el cursor al principio de la siguiente línea. |
| `\t` | Tabulador horizontal. Mueve el cursor a la siguiente posición del tabulador. |
| `\a` | Alerta. Suena la campana del sistema. |
| `\\` | Diagonal invertida. Inserta una diagonal invertida en una cadena. |
| `\"` | Comillas. Inserta unas comillas en una cadena. |

**Figura 2.2** Algunas secuencias comunes de escape.

hecho, estas comillas no se imprimen. Al utilizar la secuencia de escape **`\"`** en una cadena para que sea la salida de **`printf`**, indicamos que **`printf`** debe desplegar unas comillas.

La línea 10

```
return 0; /* indica que el programa terminó con éxito */
```

se incluye al final de toda función **`main`**. La palabra reservada **`return`** representa a uno de los diversos medios que utilizaremos para *salir de una función*. Cuando se utiliza la instrucción **`return`** al final de **`main`**, como mostramos en este caso, el valor **`0`** indica que el programa finalizó exitosamente. En el capítulo 5, explicaremos con detalle las funciones, y las razones para incluir esta instrucción serán claras. Por ahora, simplemente incluya esta instrucción en cada programa, o el compilador podría producir un mensaje de advertencia en algunos sistemas. La *llave derecha*, **`}`**, (línea12), indica el final de la función **`main`**.



### Buena práctica de programación 2.2

*Agregue un comentario a la línea que contiene la llave derecha, **`}`**, que cierra toda función, incluyendo a **`main`**.*

Dijimos que **`printf`** ocasiona que la computadora realice alguna *acción*. Cuando cualquier programa se ejecuta, éste realiza diversas acciones y toma *decisiones*. Al final de este capítulo explicamos la toma de decisiones. En el capítulo 3, explicamos a profundidad este *modelo de programación de acción/decisión*.



### Error común de programación 2.3

*Escribir en un programa el nombre de la función de salida **`printf`** como **`print`**.*

Resulta importante observar que las funciones de la biblioteca estándar como **`printf`** y **`scanf`** no forman parte del lenguaje de programación C. Por ejemplo, el compilador no puede encontrar errores de escritura en **`printf`** o **`scanf`**. Cuando el compilador compila una instrucción **`printf`**, éste sólo proporciona espacio en el programa objeto para una "llamada" a la función de biblioteca. Sin embargo, el compilador no sabe en dónde están las funciones de biblioteca; el enlazador sí lo sabe. Cuando se ejecuta el enlazador, éste localiza las funciones de biblioteca e inserta las llamadas apropiadas para dichas funciones en el programa objeto. Ahora el programa objeto está "completo" y listo para ejecutarse. De hecho, al programa enlazado con frecuencia se le conoce como *ejecutable*. Si el nombre de la función está mal escrito, es el enlazador quien detectará el error, ya que no será capaz de hacer coincidir el nombre que se encuentra en el programa en C, con el nombre de ninguna función conocida de las bibliotecas.



### Buena práctica de programación 2.3

*El último carácter que imprima cualquier función de impresión debe ser una nueva línea (**`\n`**). Esto garantiza que la función dejará al cursor de la pantalla posicionado al principio de una nueva línea. Las convenciones de esta naturaleza facilitan la reutilización de software, un objetivo clave de los ambientes de desarrollo de software.*



### Buena práctica de programación 2.4

*Establezca sangrías en el cuerpo de cada función, un nivel hacia adentro de la llave que define el cuerpo de la función (nosotros recomendamos tres espacios). Esto hará que la estructura funcional de un programa resalte, y ayudará a que los programas sean más fáciles de leer.*

```
1  /* Figura 2.3: fig02_03.c
2     Impresión de una línea mediante dos instrucciones printf */
3  #include <stdio.h>
4
5  /* la función main inicia la ejecución del programa */
6  int main()
7  {
8     printf( “Bienvenido “ );
9     printf( “a C!\n” );
10
11    return 0; /* indica que el programa terminó de con éxito */
12
13 } /* fin de la función main */
```

```
Bienvenido a C!
```

**Figura 2.3** Impresión de una línea mediante instrucciones **`printf`** separadas.



**Buena práctica de programación 2.5**

*Establezca una convención para el tamaño de la sangría que usted prefiera, y aplique de manera uniforme dicha convención. Puede utilizar la tecla de tabulación para generar la sangría, pero los saltos de tabulación pueden variar. Nosotros le recomendamos que utilice saltos de tabulación de 1/4 de pulgada, o que cuente tres espacios para formar los niveles de las sangrías.*

La función **`printf`** puede imprimir de diferentes formas el mensaje **`Bienvenido a C!`** Por ejemplo, el programa de la figura 2.3 produce la misma salida que el de la figura 2.1. Esto funciona porque cada **`printf`** continúa con la impresión a partir de donde la función **`printf`** anterior dejó de imprimir. La primera **`printf`** (línea 8) imprime **`Bienvenido`** seguido por un espacio, y la segunda **`printf`** (línea 9) comienza a imprimir en la misma línea, inmediatamente después del espacio.

Una sola **`printf`** puede imprimir varias líneas utilizando caracteres de nueva línea, como en la figura 2.4. Cada vez que aparece la secuencia de escape **`\n`** (nueva línea), la salida continúa al principio de la siguiente línea.

## 2.3 Otro programa sencillo en C: Suma de dos enteros

Nuestro siguiente programa utiliza la función **`scanf`** de la biblioteca estándar para obtener dos enteros escritos por el usuario a través del teclado, para calcular la suma de dichos valores e imprimir el resultado median-

```
/* Figura 2.4: fig02_04.c
   Impresión de múltiples líneas mediante una sola instrucción printf */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   printf( “Bienvenido\na\nC!\n” );

   return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

```
Bienvenido
a
C!
```

**Figura 2.4** Impresión en varias líneas con una sola instrucción **`printf`**.

```
1  /* Figura 2.5: fig02_05.c
2     Programa de suma */
3  #include <stdio.h>
4
5  /* la función main inicia la ejecución del programa */
6  int main()
7  {
8     int entero1; /* primer número a introducir por el usuario  */
9     int entero2; /* segundo número introducir por el usuario */
10    int suma;    /* variable en la que se almacenará la suma   */
11
12    printf( “Introduzca el primer entero\n” ); /* indicador */
13    scanf( “%d”, &entero1 );                   /* lee un entero */
14
15    printf( “Introduzca el segundo entero\n” ); /* indicador */
16    scanf( “%d”, &entero2 );                    /* lee un entero */
17
18    suma = entero1 + entero2; /* asigna el resultado a suma */
19
20    printf( “La suma es %d\n”, suma ); /* imprime la suma */
21
22    return 0; /* indica que el programa terminó con éxito */
23
24 } /* fin de la función main */
```

```
Introduzca el primer entero
45
Introduzca el segundo entero
72
La suma es 117
```

**Figura 2.5** Programa de suma.

te **`printf`**. El programa y el resultado del ejemplo aparecen en la figura 2.5. [Observe que en el diálogo de entrada/salida de la figura 2.5 resaltamos los números introducidos por el usuario.]

El comentario de las líneas 1 y 2 establece el propósito del programa. Como dijimos antes, todo programa comienza su ejecución en **`main`**. La llave izquierda, **`{`**, de la línea 7 marca el comienzo del cuerpo de **`main`**, y la llave derecha correspondiente, **`}`**, de la línea 24 manca el fin.

Las líneas 8 a 10

```
int  entero1;  /* primer número a introducir por el usuario */
int  entero2;  /* segundo número a introducir por el usuario */
int  suma;     /* variable en la que se almacenará la suma */
```

son *definiciones*. Los nombres **`entero1`**, **`entero2`**, y **`suma`** son los nombres de las *variables*. Una variable es un sitio de la memoria de la computadora en donde se puede almacenar un valor para que lo utilice un programa. Esta definición especifica que las variables **`entero1`**, **`entero2`** y **`suma`** son de tipo ***`int`***, lo cual significa que estas variables almacenan valores *enteros*, es decir, números completos como 7, −11, 0, 31914, y otros similares. Todas las variables deben declararse mediante un nombre y un tipo de dato inmediatamente después de la llave izquierda que comienza el cuerpo de **`main`**, antes de que puedan utilizarse en un programa. En C, existen otros tipos de datos además de **`int`**. Observe que hubiéramos podido combinar las definiciones anteriores en una sola instrucción de declaración de la siguiente manera:

```
int entero1, entero2, suma;
```

En C, el nombre de una variable es cualquier *identificador* válido. Un identificador es una serie de caracteres que consta de letras, dígitos y guiones bajos (_), y que no comienza con un dígito. Un identificador pue-

de tener cualquier longitud, sin embargo, los compiladores de C sólo requieren reconocer los primeros 31 caracteres, de acuerdo con el ANSI C estándar. C es *sensible a mayúsculas y minúsculas*, de tal forma que **`a1`** y **`A1`** son identificadores diferentes.



### Error común de programación 2.4

*Utilizar una letra mayúscula cuando debe utilizarse una minúscula (por ejemplo, escribir* **`Main`** *en lugar de* **`main`***).*



### Tip de portabilidad 2.1

*Utilice identificadores de 31 caracteres o menos. Esto le ayudará a garantizar la portabilidad y puede evitar algunos problemas sutiles de programación.*



### Buena práctica de programación 2.6

*Elegir nombres de variables que tengan significado le ayuda a escribir programas "autodocumentados"; es decir, necesitará menos comentarios.*



### Buena práctica de programación 2.7

*La primera letra de un identificador utilizado como un nombre de variable sencillo debe ser minúscula. Más adelante asignaremos un significado especial a los identificadores que comienzan con una letra mayúscula, y a los identificadores que utilizan todas sus letras en mayúsculas.*



### Buena práctica de programación 2.8

*Los nombres de variables con muchas palabras pueden ayudarle a escribir un programa más legible. Evite juntar palabras diferentes como* **`comisionestotales`***; mejor utilice las palabras separadas por un guión bajo como en* **`comisiones_totales`***, o, si desea juntar las palabras, comience cada una con letras mayúsculas como en* **`ComisionesTotales`***. Este último estilo es preferible.*

La declaración de variables debe colocarse después de la llave izquierda de una función y antes de *cualquier* instrucción ejecutable. Por ejemplo, en el programa de la figura 2.5, insertar las declaraciones después del primer **`printf`** ocasionaría un error de sintaxis. Sucede un *error de sintaxis* cuando el compilador no puede reconocer una instrucción. El compilador por lo general envía un mensaje de error para ayudar al programador a localizar y a arreglar la instrucción incorrecta. Los errores de sintaxis son violaciones del lenguaje. A estos errores también se les conoce como *errores de compilación*, o *errores en tiempo de compilación*.



### Error común de programación 2.5

*Colocar las declaraciones de variables entre instrucciones ejecutables, ocasiona errores de sintaxis.*



### Buena práctica de programación 2.9

*Separe las declaraciones y las instrucciones ejecutables de una función mediante una línea en blanco, para resaltar donde terminan las declaraciones y donde comienzan las instrucciones ejecutables.*

La línea 12

```
printf( "Introduzca el primer entero\n" ); /*indicador */
```

imprime en la pantalla las palabras **`Introduzca el primer entero`**, y posiciona el cursor a principio de la siguiente línea. A este mensaje se le llama indicador porque le indica al usuario que realice una acción específica.

La siguiente instrucción

```
scanf( "%d", &entero1 ); /* lee un entero */
```

utiliza **`scanf`** para obtener un valor por parte del usuario. La función **`scanf`** toma la información de entrada desde la entrada estándar que, por lo general, es el teclado. Esta **`scanf`** tiene dos argumentos, **`"%d"`** y **`&entero1`**. El primer argumento, la *cadena de control de formato*, indica el tipo de dato que debe introducir el usuario. El *especificador de conversión*, **`%d`**, indica que el dato debe ser un entero (la letra **`d`** significa "entero decimal"). En este contexto, **`scanf`** (y también **`printf`**, como veremos más adelante) trata al **`%`** como un carácter especial que comienza un especificador de conversión. El segundo argumento de **`scanf`** comienza con un amperson (**&**), conocido en C como *operador de dirección*, seguido del nombre de una variable. El amperson, cuando se combina con el nombre de una variable, le indica a **`scanf`** la ubicación en memoria de la va-

riable **entero1**. La computadora después almacena el valor de **entero1** en esa ubicación. El uso del amperson (**&**) con frecuencia es confuso para los programadores principiantes o para la gente que ha programado en otros lenguajes que no requieren esta notación. Por el momento, sólo recuerde que debe colocar un amperson antes de cada variable en cualquier instrucción **scanf**. En los capítulos 6 y 7 explicamos algunas excepciones a esta regla. El uso del amperson será claro después de que estudiemos los apuntadores en el capítulo 7.

### Buena práctica de programación 2.10

*Coloque un espacio después de cada coma (**,**), para hacer que los programas sean más legibles.*

Cuando la computadora ejecuta la instrucción **scanf** anterior, ésta espera a que el usuario introduzca un valor para la variable **entero1**. El usuario responde escribiendo un entero y después oprimiendo la *tecla de retorno* (algunas veces llamada tecla *Entrar*), para enviar el número a la computadora. Después, la computadora asigna este número, o *valor*, a la variable **entero1**. Cualquier referencia posterior a **entero1** en el programa utilizará este mismo valor. Las funciones **printf** y **scanf** facilitan la interacción entre el usuario y la computadora. Debido a que esta interacción parece un diálogo, con frecuencia se le llama *computación conversacional* o *computación interactiva*.

La línea 15

```
printf( "Introduzca el segundo entero\n" ); /*indicador */
```

despliega en la pantalla el mensaje **Introduzca el segundo entero**, y después coloca el cursor al principio de la siguiente línea. La instrucción **printf** también indica al usuario que realice esa acción.

La instrucción

```
scanf( "%d", &entero2 ); /*lee un entero */
```

obtiene un valor para la variable **entero2** por parte del usuario. La *instrucción de asignación* de la línea 18

```
suma = entero1 + entero2; /* asigna el resultado a suma */
```

calcula la suma de las variables **entero1** y **entero2**, y asigna el resultado a la variable **suma** mediante el *operador de asignación* **=**. La instrucción se lee como, "**suma** *obtiene* el valor de **entero1 + entero2**". La mayoría de los cálculos se realizan en instrucciones de asignación. El operador = y el operador + se conocen como *operadores binarios*, ya que cada uno de ellos tiene dos *operandos*. En el caso del operador +, los dos operandos son **entero1** y **entero2**. En el caso del operador =, los dos operandos son **suma** y el valor de la expresión **entero1 + entero2**.



### Buena práctica de programación 2.11

*Coloque espacios a cada lado de un operador binario. Esto hace que el operador resalte, y hace más claro el programa.*



### Error común de programación 2.6

*Los cálculos en las instrucciones de asignación deben estar a la derecha del operador =. Colocar los cálculos a la izquierda de un operador de asignación, es un error de sintaxis.*

La línea 20

```
printf( "La suma es %d\n", suma ); /* imprime suma */
```

llama a la función **printf** para que despliegue en la pantalla las palabras **La suma es**, seguidas del valor numérico de la variable **suma**. Esta función **printf** tiene dos argumentos, **"La suma es %d\n"** y **suma**. El primer argumento es la cadena de control de formato. Ésta contiene algunos caracteres literales que se desplegarán, y contiene el especificador de conversión **%d**, que indica que se imprimirá un entero. El segundo argumento especifica el valor que se imprimirá. Observe que el especificador de conversión para un entero es el mismo tanto en **printf** como en **scanf**. Es el mismo caso para la mayoría de los tipos de datos en C.

Los cálculos también pueden realizarse en instrucciones **printf**. Nosotros hubiéramos podido combinar las dos instrucciones anteriores en la instrucción

```
printf( "La suma es %d\n, entero1 + entero2 );
```

La línea 22

```
return 0; /* indica que el programa terminó con éxito */
```

pasa el valor **0** de regreso al ambiente del sistema operativo en el que el programa se está ejecutando. Esto indica al sistema operativo que el programa se ejecutó con éxito. Para obtener información sobre cómo reportar una falla del programa, vea los manuales de su sistema operativo en particular.

La llave derecha, **}**, de la línea 24 indica que se llegó al final de la función **main**.



**Error común de programación 2.7**

*Olvidar una o ambas comillas alrededor de la cadena de control de formato en una instrucción **printf** o **scanf**.*



**Error común de programación 2.8**

*Olvidar el **%** en una especificación de conversión en la cadena de control de formato de una instrucción **printf** o **scanf**.*



**Error común de programación 2.9**

*Colocar una secuencia de escape como **\n** fuera de la cadena de control de formato de una instrucción **printf** o **scanf**.*



**Error común de programación 2.10**

*Olvidar incluir las expresiones cuyos valores van a imprimirse en una instrucción **printf** que contiene especificadores de conversión.*



**Error común de programación 2.11**

*No proporcionar a una cadena de control de formato, correspondiente a una instrucción **printf**, un especificador de conversión, cuando se necesita uno para imprimir el valor de una expresión.*



**Error común de programación 2.12**

*Colocar dentro de una cadena de control de formato la coma que se supone debe separar la cadena de control de formato de las expresiones a imprimirse.*



**Error común de programación 2.13**

*Olvidar colocar un amperson antes de una variable correspondiente a una instrucción **scanf**, cuando, de hecho, debe ser precedida por uno.*

En muchos sistemas, el error de ejecución anterior ocasiona una “falla de segmentación” o “violación de acceso”. Dicho error ocurre cuando algún usuario del sistema intenta acceder a una parte de la memoria de la computadora, a la que no tiene privilegios de acceso. En el capítulo 7, explicaremos la causa precisa de este error.



**Error común de programación 2.14**

*Colocar un amperson antes de una variable incluida en una instrucción **printf**, cuando, de hecho, no debe ser precedida por uno.*

## 2.4 Conceptos de memoria

Los nombres de variables tales como **entero1**, **entero2** y **suma** en realidad corresponden a lugares en la memoria de la computadora. Toda variable tiene un *nombre*, un *tipo* y un *valor*.

En el programa de suma de la figura 2.5, cuando la instrucción (línea 13)

```
scanf( “%d”, &entero1 ); /* lee un entero */
```

se ejecuta, el valor escrito por el usuario se coloca en un lugar de la memoria al que se le ha asignado el nombre de **entero1**. Suponga que el usuario escribe el número **45** como el valor para **entero1**. La computadora colocará **45** en el lugar de **entero1**, como muestra la figura 2.6.

Siempre que un valor se coloca en una posición de memoria, dicho valor reemplaza al valor anterior de esa ubicación. Debido a que la información anterior se destruye, el proceso de lectura de información en una ubicación de memoria se conoce como *lectura destructiva*.




**Figura 2.6** Ubicación de memoria que muestra el nombre y el valor de una variable.

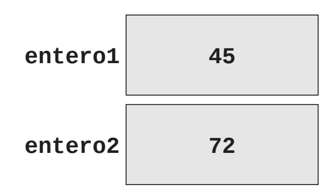



**Figura 2.7** Ubicaciones de memoria después de introducir ambas variables.

Volviendo nuevamente a nuestro programa de suma, cuando la instrucción (línea 16)

```
scanf( “%d”, &entero2 ); /* lee un entero */
```

se ejecuta, suponga que el usuario escribe el valor **72**. Este valor se coloca en una ubicación llamada **entero2**, y la memoria luce como en la figura 2.7. Observe que estas ubicaciones no necesariamente están adyacentes en memoria.

Una vez que el programa obtuvo los valores de **entero1** y **entero2**, éste suma los valores y coloca el resultado en la variable **suma**. La instrucción (línea 18)

```
suma = entero1 + entero2; /* asigna el resultado a suma */
```

que realiza la suma también involucra una lectura destructiva. Esto ocurre cuando la suma calculada de **entero1** y **entero2** se coloca en la ubicación de **suma** (destruyendo el valor que pudo haber estado en **suma**). Después de que se calcula la suma, la memoria luce como en la figura 2.8. Observe que los valores de **entero1** y **entero2** aparecen exactamente como estaban antes de que se utilizaran para calcular la **suma**. Estos valores se utilizaron, pero no se destruyeron, cuando la computadora realizó el cálculo. Por lo tanto, cuando se lee un valor desde una posición de memoria, el proceso se conoce como *lectura no destructiva*.

## 2.5 Aritmética en C

La mayoría de los programas en C realizan cálculos aritméticos. Los *operadores aritméticos* de C aparecen en la figura 2.9. Observe que se utilizan varios símbolos especiales que no se emplean en álgebra. El *asterisco* **(*)** indica una multiplicación y el *signo de porcentaje* **(%)** es el operador *módulo*, el cual explicaremos más adelante. En álgebra, si queremos multiplicar *a* por *b*, simplemente colocamos estas letras, que corresponden al nombre de las variables, una junto a la otra, es decir, *ab*. Sin embargo, en C, si hiciéramos esto, **ab** se interpre-

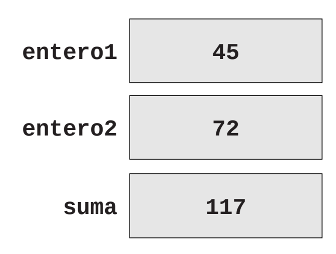



**Figura 2.8** Ubicaciones de memoria después de un cálculo.

| Operación en C | Operador aritmético | Expresión algebraica | Expresión en C |
|---|---|---|---|
| Suma | **+** | $f + 7$ | **f + 7** |
| Resta | **-** | $p - c$ | **p - c** |
| Multiplicación | ***** | $bm$ | **b * m** |
| División | **/** | $x / y$ o $\frac{x}{y}$ o $x \div y$ | **x / y** |
| Módulo | **%** | $r \bmod s$ | **r % s** |

**Figura 2.9** Operadores aritméticos.

taría como un solo nombre (o identificador) de dos letras. Por lo tanto, C (y otros lenguajes de programación) requiere que el usuario denote explícitamente la multiplicación mediante el operador `*`, como `a*b`.

Todos los operadores aritméticos son operadores binarios. Por ejemplo, la expresión `3 + 7` contiene el operador binario `+` y los operandos `3` y `7`.

La *división entera* arroja un resultado entero. Por ejemplo, `7 / 4` da como resultado `1`, y la expresión `17 / 5` da como resultado `3`. C proporciona el operador módulo, `%`, el cual arroja el residuo de una división entera. El operador módulo es un operador entero que puede utilizarse sólo con operandos enteros. La expresión `x%y` arroja el residuo, después de que `x` se divide entre `y`. Por lo tanto, `7%4` arroja `3`, y `17%5` arroja `2`. Explicaremos muchas aplicaciones interesantes del operador módulo.

### Error común de programación 2.15

*La división entre cero por lo general no está definida en los sistemas de cómputo, y da como resultado un error fatal, es decir, un error que ocasiona que el programa termine de inmediato, sin que haya finalizado con éxito su trabajo. Los errores no fatales permiten a los programas ejecutarse totalmente, pero con frecuencia producen resultados incorrectos.*

Las expresiones aritméticas en C deben escribirse en forma de *línea recta* para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "`a` dividida entre `b`" debe escribirse como `a/b`, para que todos los operadores y operandos aparezcan en línea recta. En general, los compiladores no aceptan la notación algebráica:

$$\frac{a}{b}$$

aunque existen algunos paquetes especiales de software que permiten una notación más natural para expresiones matemáticas complejas.

Los paréntesis se utilizan para agrupar términos en expresiones de C, casi de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por `b + c` escribimos:

```
a * (b + c)
```

C evalúa las expresiones aritméticas en una secuencia precisa, determinada por las siguientes *reglas de precedencia de operadores*, las cuales generalmente son las mismas que las que aplicamos en álgebra:

1. Las operaciones de multiplicación, división y módulo se aplican primero. En una expresión que contiene varias operaciones de multiplicación, división y módulo, la evaluación se realiza de izquierda a derecha. Se dice que la multiplicación, la división y el residuo tienen el mismo nivel de precedencia.
2. Las operaciones de suma y resta se aplican después. Si una expresión contiene varias operaciones de suma y resta, la evaluación se realiza de izquierda a derecha. La suma y la resta también tienen el mismo nivel de precedencia, el cual es menor que el de la precedencia de los operadores de multiplicación, división y módulo.

Las reglas de precedencia de operadores son una guía que permite a C evaluar expresiones en el orden correcto. Cuando decimos que la evaluación se realiza de izquierda a derecha, nos referimos a la *asociatividad* de los operadores. Veremos que algunos operadores asocian de derecha a izquierda. La figura 2.10 resume estas reglas de precedencia de operadores.

| Operador(es) | Operación(es) | Orden de evaluación (precedencia) |
| --- | --- | --- |
| `*` | Multiplicación | Se evalúan primero. Si hay muchas, se evalúan de izquierda a derecha. |
| `/` | División | |
| `%` | Módulo | |
| `+` | Suma | Se evalúan después. Si hay muchas, se evalúan de izquierda a derecha. |
| `-` | Resta | |

**Figura 2.10** Precedencia de operadores aritméticos.

Ahora consideremos varias expresiones para aclarar las reglas de precedencia de operadores. Cada ejemplo muestra una expresión algebraica y su equivalente en C.

El siguiente ejemplo calcula la media aritmética (promedio) de cinco términos:

Álgebra: $m = \dfrac{a + b + c + d + e}{5}$

```
C:     m = ( a + b + c + d + e ) / 5;
```

Los paréntesis son necesarios para agrupar las sumas, ya que la división tiene un nivel de precedencia más alto que la suma. La cantidad completa **`( a + b + c + d + e )`** debe dividirse entre **`5`**. Si por error los paréntesis se omiten, obtenemos **`a + b + c + d + e/5`**, lo que se evalúa incorrectamente como

$$a + b + c + d + \frac{e}{5}$$

El siguiente ejemplo muestra la ecuación de una línea recta:

Álgebra: $y + mx + b$

```
C:      y = m * x + b;
```

En este caso no se requieren paréntesis. La multiplicación se evalúa primero, ya que ésta tiene un nivel de precedencia mayor que la suma.

El siguiente ejemplo contiene las operaciones de módulo **`(%)`**, multiplicación, división, suma, resta y de asignación:

Álgebra: $z = pr\%q+w/x–y$

```
C:      z =  p  *  r  %  q  +  w  /  x  –  y
          6     1     2     4     3     5
```

Los números que se encuentran circulados y que aparecen debajo de la instrucción indican el orden en el que C evalúa los operadores. La multiplicación, el módulo y la división se evalúan primero, en orden de izquierda a derecha (es decir, asocian de izquierda a derecha), ya que tiene un nivel de precedencia mayor que la suma y la resta. Después se evalúan la suma y la resta. Éstas también se evalúan de izquierda a derecha.

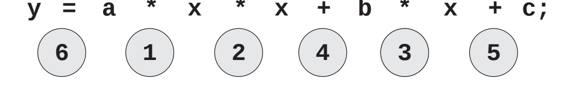
No todas las expresiones con varios pares de paréntesis contienen paréntesis anidados. La expresión

```
a * ( b + c ) + c * ( d + e )
```

no contiene paréntesis anidados. En cambio, se dice que los paréntesis tienen el mismo nivel de precedencia.

Para comprender mejor las reglas de precedencia de operadores, veamos cómo es que C evalúa un polinomio de segundo grado.

```
y = a * x * x + b * x + c;
```



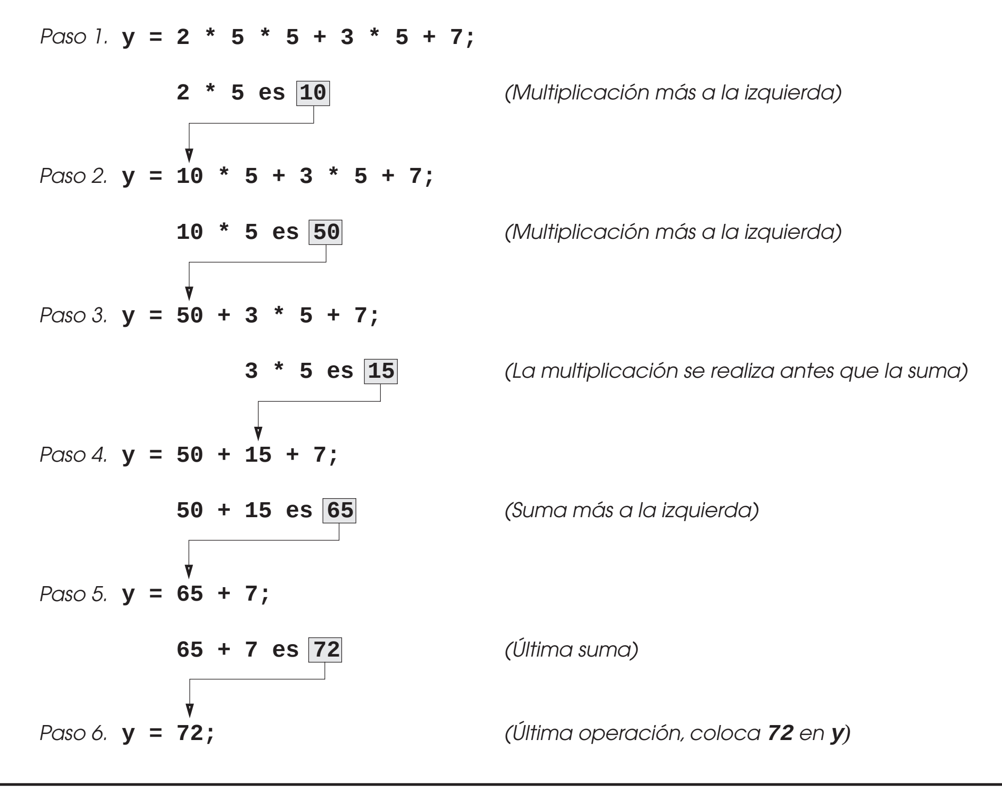



**Figura 2.11** Orden en el que se evalúa un polinomio de segundo grado.

Los números circulados que aparecen bajo la instrucción indican el orden en el que C realiza las operaciones. En C no existe un operador aritmético para la exponenciación, por lo que representamos $x^2$ como **x * x**. La Biblioteca Estándar de C incluye la función **pow** (“potencia”), para llevar a cabo exponenciaciones. Debido a algunos detalles sutiles relacionados con los tipos de datos que requiere **pow**, posponemos la explicación de dicha función para el capítulo 4.

Considere que **a=2**, **b=3**, **c=7**, y **x=5**. La figura 2.11 muestra cómo se evalúa el polinomio de segundo grado anterior.

## 2.6 Toma de decisiones: Operadores de igualdad y de relación

Las instrucciones ejecutables de C realizan *acciones* (como cálculos, o entradas o salidas de datos), o toman *decisiones* (pronto veremos varios ejemplos de esto). Como ejemplo, podríamos tomar una decisión con un programa, para determinar si la calificación que una persona obtuvo en un examen es mayor o igual que 60, y si es así, imprimir el mensaje “¡Felicidades! aprobó el examen”. Esta sección presenta una versión sencilla de la *instrucción* ***if*** de C, la cual permite a un programa tomar una decisión, basándose en la verdad o falsedad de una instrucción de hechos, llamada *condición*. Si se cumple la condición, es decir, la condición es *verdadera*, se ejecuta la instrucción en el cuerpo de la instrucción **if**. Si la condición no se cumple, es decir, la condición es *falsa*, no se ejecuta la instrucción en el cuerpo de la estructura. Ya sea que la instrucción se ejecute o no, una vez que se completa la instrucción **if**, la ejecución continúa con la siguiente instrucción después de **if**.

Las condiciones en instrucciones **if** se forman utilizando los *operadores de igualdad* y *de relación* que aparecen en la figura 2.12. Todos los operadores de relación tienen el mismo nivel de precedencia, y se asocian de izquierda a derecha. Los operadores de igualdad tienen un nivel de precedencia más bajo que los operadores de relación, y ellos también se asocian de izquierda a derecha. [*Nota:* En C, una condición puede ser cualquier expresión que genere un valor cero (falso) o uno diferente de cero (verdadero). A lo largo del libro veremos muchas aplicaciones de esto.]

| Operador algebraico estándar de igualdad o de relación | Operador de igualdad o de relación en C | Ejemplo de una condición en C | Significado de la condición en C |
|---|---|---|---|
| *Operadores de igualdad* | | | |
| = | `==` | `x == y` | **x** es igual que **y** |
| ≠ | `!=` | `x != y` | **x** no es igual que **y** |
| *Operadores de relación* | | | |
| > | `>` | `x > y` | **x** es mayor que **y** |
| < | `<` | `x < y` | **x** es menor que **y** |
| ≥ | `>=` | `x ≥= y` | **x** es mayor o igual que **y** |
| ≤ | `<=` | `x ≤= y` | **x** es menor o igual que **y** |

**Figura 2.12** Operadores de igualdad y de relación.



**Error común de programación 2.16**

*Ocurrirá un error de sintaxis si los dos símbolos de cualquiera de los operadores* **==**, **!=**, **>=** *y* **<=** *aparecen separados por un espacio.*



**Error común de programación 2.17**

*Ocurrirá un error de sintaxis si se invierten los símbolos de cualquiera de los operadores* **!=**, **>=** *y* **<=**, *como* **=!**, **=>**, **=<**, *respectivamente.*



**Error común de programación 2.18**

*Confundir el operador de igualdad* **==** *con el operador de asignación* **=**.

Para evitar esta confusión, el operador de igualdad debe leerse como “doble igualdad”, y el operador de asignación como “obtiene”. Como veremos pronto, confundir estos operadores no necesariamente ocasiona errores de sintaxis fáciles de reconocer, pero puede causar errores lógicos extremadamente sutiles.



**Error común de programación 2.19**

*Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho después de la condición de una instrucción* ***if***.

La figura 2.13 utiliza seis instrucciones **if** para comparar dos números introducidos por el usuario. Si se satisface la condición en cualquiera de estas instrucciones **if**, se ejecutará la instrucción **printf** asociada con ese **if**. En la figura aparecen el programa y los resultados de tres ejecuciones de ejemplo.

```
/* Figura 2.13: fig02_13.c
   Uso de instrucciones if, operadores
   de relación, y operadores de igualdad */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
main()
{
   int num1; /* primer número que lee el usuario  */
   int num2; /* segundo número que lee el usuario */

   printf( “Introduzca dos enteros, y le dire\n” );
   printf( “las relaciones que satisfacen: “ );

```

**Figura 2.13** Uso de los operadores de igualdad y de relación. (Parte 1 de 2.)

```
15      scanf( "%d%d", &num1, &num2 ); /* lectura de los enteros */
16
17      if ( num1 == num2 ) {
18         printf( "%d es igual que  %d\n", num1, num2 );
19      } /* fin de if */
20
21      if ( num1 != num2 ) {
22         printf( "%d no es igual que %d\n", num1, num2 );
23      } /* fin de if */
24
25      if ( num1 < num2 ) {
26         printf( "%d es menor que %d\n", num1, num2 );
27      } /* fin de if */
28
29      if ( num1 > num2 ) {
30         printf( "%d es mayor que %d\n", num1, num2 );
31      } /* fin de if */
32
33      if ( num1 <= num2 ) {
34         printf( "%d es menor o igual que %d\n", num1, num2 );
35      } /* fin de if */
36
37      if ( num1 >= num2 ) {
38         printf( "%d es mayor o igual que %d\n", num1, num2 );
39      } /* fin de if */
40
41      return 0; /* indica que el programa terminó con éxito */
42
43   } /* fin de la función main */
```

```
Introduzca dos enteros, y le dire
las relaciones que satisfacen: 3 7
3 no es igual que 7
3 es menor que 7
3 es menor o igual que 7
```

```
Introduzca dos enteros, y le dire
las relaciones que satisfacen: 22 12
22 no es igual que 12
22 es mayor que 12
22 es mayor o igual que 12
```

```
Introduzca dos enteros, y le dire
las relaciones que satisfacen: 7 7
7 es igual que  7
7 es menor o igual que 7
7 es mayor o igual que 7
```

**Figura 2.13** Uso de los operadores de igualdad y de relación. (Parte 2 de 2.)

Observe que el programa de la figura 2.13 utiliza la función **`scanf`** (línea 15) para introducir dos números. Cada especificador de conversión tiene un argumento correspondiente, en el que se almacenará un valor. El primer **%d** convierte un valor que se almacenará en la variable **`num1`**, y el segundo **%d** convierte un valor que se almace-

| Operadores | | | | Asociatividad |
|---|---|---|---|---|
| `*` | `/` | `%` | | izquierda a derecha |
| `+` | `–` | | | izquierda a derecha |
| `<` | `<=` | `>` | `>=` | izquierda a derecha |
| `=` | `= !` | | | izquierda a derecha |
| `=` | | | | derecha a izquierda |

**Figura 2.14** Precedencia y asociatividad de los operadores que hemos explicado hasta el momento.

nará en la variable **`num2`**. Colocar sangrías a lo largo del cuerpo de cada instrucción **`if`**, y colocar líneas en blanco arriba y debajo de cada una de ellas mejora la legibilidad del programa. Además, observe que cada instrucción **`if`** de la figura 2.13 tiene una sola instrucción en su cuerpo. En el capítulo 3, mostramos cómo especificar instrucciones **`if`** con cuerpos formados por múltiples instrucciones.



**Buena práctica de programación 2.12**

*Coloque sangrías en el cuerpo de una instrucción **`if`**.*



**Buena práctica de programación 2.13**

*Coloque una línea en blanco antes y después de cada instrucción **`if`**, para mejorar la legibilidad del programa.*



**Buena práctica de programación 2.14**

*Aunque está permitido, en un programa no debe haber más de una instrucción por línea.*



**Error común de programación 2.20**

*Colocar comas (cuando no son necesarias) entre especificadores de conversión en la cadena de control de formato correspondiente a una instrucción **`scanf`**.*

El comentario (líneas 1 a 3) de la figura 2.13 está separado en tres líneas. En los programas en C, los *espacios blancos* como tabuladores, nuevas líneas y espacios, por lo general son ignorados. Por lo tanto, las instrucciones y comentarios deben extenderse en varias líneas. Sin embargo, no es correcto separar identificadores.



**Buena práctica de programación 2.15**

*Una instrucción larga puede distribuirse en varias líneas. Si una instrucción debe separarse a lo largo de varias líneas, elija límites que tengan sentido (como después de una coma, en una lista separada por comas). Si una instrucción se divide en dos o más líneas, coloque sangrías en todas las líneas subsiguientes.*

La figura 2.14 lista la precedencia de los operadores que presentamos en este capítulo. Los operadores aparecen de arriba abajo en orden decreciente de precedencia. Observe que el signo de igualdad también es un operador. Todos estos operadores, con excepción del de asignación **`=`**, asocian de izquierda a derecha. El operador de asignación (**`=`**) asocia de derecha a izquierda.



**Buena práctica de programación 2.16**

*Revise la tabla de precedencia de operadores, cuando escriba expresiones que contengan muchos operadores. Confirme que los operadores de la expresión se aplican en el orden correcto. Si no está seguro del orden de evaluación de una expresión compleja, utilice paréntesis para agrupar expresiones. Asegúrese de recordar que algunos de los operadores de C, como el de asignación (**`=`**), asocian de derecha a izquierda, y no de izquierda a derecha.*

Algunas de las palabras que hemos utilizado en los programas en C de este capítulo, en particular **`int`**, **`return`** e **`if`**, son *palabras clave* o *palabras reservadas* del lenguaje. Las palabras reservadas de C aparecen en la figura 2.15. Estas palabras tienen un significado especial para el compilador de C, por lo que el programador debe tener cuidado de no utilizar estas palabras como identificadores, tales como nombres de variables. En este libro, explicaremos todas estas palabras reservadas.

| Palabras reservadas | | | |
|---|---|---|---|
| `auto` | `double` | `int` | `struct` |
| `break` | `else` | `long` | `switch` |
| `case` | `enum` | `register` | `typedef` |
| `char` | `extern` | `return` | `union` |
| `const` | `float` | `short` | `unsigned` |
| `continue` | `for` | `signed` | `void` |
| `default` | `goto` | `sizeof` | `volatile` |
| `do` | `if` | `static` | `while` |

**Figura 2.15** Palabras reservadas de C.

En este capítulo introdujimos muchas características importantes del lenguaje de programación C, que incluyen la impresión de datos en pantalla, la introducción de datos por parte del usuario, la realización de cálculos y la toma de decisiones. En el siguiente capítulo, fortaleceremos estas técnicas, conforme presentemos la programación estructurada. Estudiaremos cómo especificar el orden en el que se ejecutan las instrucciones; a esto se le conoce como flujo de control.

## *RESUMEN*

- Los comentarios comienzan con **/*** y terminan con ***/**. Los programadores insertan comentarios para documentar sus programas y para mejorar su legibilidad. Los comentarios no ocasionan acción alguna cuando se ejecuta el programa.
- La directiva del preprocesador **#include<stdio.h>** le indica al compilador que incluya en el programa el encabezado estándar de entrada/salida. Este archivo contiene información que el compilador utiliza para verificar la precisión de las llamadas a funciones de entrada y salida, como **scanf** y **printf**.
- Los programas en C consisten en funciones, una de las cuales debe ser **main**. Todo programa en C comienza su ejecución en la función **main**.
- La función **printf** puede utilizarse para imprimir una cadena que se encuentra entre comillas, y para imprimir los valores de expresiones. Cuando se imprime un valor entero, el primer argumento de la función **printf** (la cadena de control de formato) contiene el especificador de conversión **%d** y cualquier otro carácter a imprimir; el segundo argumento es la expresión cuyo valor se imprimirá. Si se va a imprimir más de un entero, la cadena de control de formato contiene un **%d** para cada entero, y los argumentos separados por comas que siguen a la cadena de control de formato contienen las expresiones cuyos valores se imprimirán.
- La función **scanf** obtiene valores que el usuario normalmente introduce por medio del teclado. Su primer argumento es la cadena de control de formato que le indica a la computadora qué tipo de dato debe introducir el usuario. El especificador de conversión, **%d**, indica que el dato debe ser un entero. Cada uno de los argumentos restantes corresponden a uno de los especificadores de conversión de la cadena de control de formato. En general, todo nombre de variable es precedido por un amperson (**&**), llamado operador de dirección. El amperson, cuando se combina con el nombre de una variable, le indica a la computadora la posición de memoria en donde se almacenará el valor. Después la computadora almacena el valor en esa posición.
- Todas las variables deben declararse antes de que puedan utilizarse en un programa.
- Un nombre de variable es cualquier identificador válido. Un identificador es una serie de caracteres compuestos por letras, dígitos y guiones bajos (_). Los identificadores no deben comenzar con un dígito. Los identificadores pueden tener cualquier longitud, sin embargo, sólo los primeros 31 dígitos son importantes.
- C es sensible a mayúsculas y minúsculas.
- La mayoría de los cálculos se realizan en instrucciones de asignación.
- Toda variable almacenada en la memoria de la computadora tiene un nombre, un valor y un tipo.
- Siempre que un nuevo valor se coloque en una posición de memoria, éste reemplaza al valor anterior de esa posición. Debido a que la información anterior se destruye, al proceso de leer información en una posición de memoria se le conoce como lectura destructiva.

- Al proceso de lectura desde una posición de memoria se le conoce como lectura no destructiva.
- Las expresiones aritméticas deben escribirse en forma de línea recta, para facilitar la introducción de programas a la computadora.
- El compilador evalúa expresiones aritméticas en una secuencia precisa, determinada por las reglas de precedencia y de asociatividad de operadores.
- La instrucción **`if`** permite al programador tomar decisiones cuando se cumple cierta condición.
- Si la condición es verdadera, entonces se ejecuta la instrucción en el cuerpo de **`if`**. Si la condición es falsa, se salta la instrucción del cuerpo.
- Por lo general, las condiciones en instrucciones **`if`** se forman utilizando operadores de igualdad o de relación. El resultado de utilizar estos operadores siempre es la simple observación de “verdadero” o “falso”. Observe que las condiciones pueden ser expresiones que generen un valor cero (falso), o uno diferente de cero (verdadero).

## TERMINOLOGÍA

acción
amperson (**&**)
argumento
asociatividad de derecha a izquierda
asociatividad de izquierda a derecha
asociatividad de operadores
asterisco (**`*`**)
biblioteca Estándar de C
bloque
cadena
cadena de caracteres
cadena de control
cadena de control de formato
carácter de escape
carácter de escape diagonal invertida (****)
carácter de nueva línea (**`\n`**)
carácter espacio en blanco
comentario
computación conversacional
computación interactiva
condición
cuerpo de una función
decisión
declaración
división entera
división entre cero
encabezado estándar de entrada/salida
entero
error de compilación
error de sintaxis
error en tiempo de compilación
error fatal
error no fatal
especificador de conversión
especificador de conversión **`%d`**
falso
flujo de control
forma de línea recta
función
función printf
función scanf
guión bajo (_)
identificador
indicador
instrucción
instrucción de asignación
instrucción de control **`if`**
**`int`**
lectura no destructiva
literal
llaves **`{}`**
main
memoria
mensaje
modelo de acción/decisión
nombre
nombre de variable
operador
operador de asignación (**`=`**)
operador de asignación signo de igual (**`=`**)
operador de dirección
operador de multiplicación (**`*`**)
operador módulo (**`%`**)
operadores aritméticos
operadores binarios
operadores de igualdad
    **`==`** “es igual que”
    **`!=`** “no es igual que”
operadores de relación
    **`>`** “es mayor que”
    **`<`** “es menor que”
    **`>=`** “es mayor o igual que”
    **`<=`** “es menor o igual que”
operando
palabras clave
palabras reservadas
palabras reservadas de C
paréntesis **`( )`**
paréntesis anidados
posición, ubicación
precedencia
preprocesador de C
programación estructurada
reglas de precedencia de operadores
sangría
secuencia de escape
sensible a mayúsculas y minúsculas
signo de porcentaje (**`%`**) para iniciar un especificador de conversión
**`stdio.h`**
tecla de retorno
tecla Entrar
terminador de instrucción (**`;`**)
terminador de instrucción punto y coma (**`;`**)
tipo de variable
toma de decisiones
ubicación (o posición) de memoria
valor
valor cero (falso)
valor de variable
valor diferente de cero (verdadero)
variable
verdadero

## ERRORES COMUNES DE PROGRAMACIÓN

**2.1** Olvidar finalizar un comentario con **`/*`**.
**2.2** Comenzar un comentario con los caracteres **`*/`**, o finalizarlo con **`/*`**.

**2.3** Escribir en un programa el nombre de la función de salida `printf` como `print`.

**2.4** Utilizar una letra mayúscula cuando debe utilizarse una minúscula (por ejemplo, escribir `Main` en lugar de `main`).

**2.5** Colocar las declaraciones de variables entre instrucciones ejecutables, ocasiona errores de sintaxis.

**2.6** Los cálculos en las instrucciones de asignación deben estar a la derecha del operador =. Colocar los cálculos a la izquierda de un operador de asignación, es un error de sintaxis.

**2.7** Olvidar una o ambas comillas alrededor de la cadena de control de formato en una instrucción `printf` o `scanf`.

**2.8** Olvidar el `%` en una especificación de conversión en la cadena de control de formato de una instrucción `printf` o `scanf`.

**2.9** Colocar una secuencia de escape como `\n` fuera de la cadena de control de formato de una instrucción `printf` o `scanf`.

**2.10** Olvidar incluir las expresiones cuyos valores van a imprimirse en una instrucción `printf` que contiene especificadores de conversión.

**2.11** No proporcionar a una cadena de control de formato, correspondiente a una instrucción `printf`, un especificador de conversión, cuando se necesita uno para imprimir el valor de una expresión.

**2.12** Colocar dentro de una cadena de control de formato la coma que se supone debe separar la cadena de control de formato de las expresiones a imprimirse.

**2.13** Olvidar colocar un amperson antes de una variable correspondiente a una instrucción `scanf`, cuando, de hecho, debe ser precedida por uno.

**2.14** Colocar un amperson antes de una variable incluida en una instrucción `printf`, cuando, de hecho, no debe ser precedida por uno.

**2.15** La división entre cero por lo general no está definida en los sistemas de cómputo, y da como resultado un error fatal, es decir, un error que ocasiona que el programa termine de inmediato, sin que haya finalizado con éxito su trabajo. Los errores no fatales permiten a los programas ejecutarse totalmente, pero con frecuencia producen resultados incorrectos.

**2.16** Ocurrirá un error de sintaxis si los dos símbolos de cualquiera de los operadores `==`, `!=`, `>=` y `<=` aparecen separados por un espacio.

**2.17** Ocurrirá un error de sintaxis si se invierten los símbolos de cualquiera de los operadores `!=`, `>=` y `<=`, como `=!`, `=>`, `=<`, respectivamente.

**2.18** Confundir el operador de igualdad `==` con el operador de asignación `=`.

**2.19** Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho después de la condición de una instrucción `if`.

**2.20** Colocar comas (cuando no son necesarias) entre especificadores de conversión en la cadena de control de formato correspondiente a una instrucción `scanf`.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**2.1** Toda función debe ser precedida por un comentario que describa el propósito de la función.

**2.2** Agregue un comentario a la línea que contiene la llave derecha, `}`, que cierra toda función, incluyendo a `main`.

**2.3** El último carácter que imprima cualquier función de impresión debe ser una nueva línea (`\n`). Esto garantiza que la función dejará al cursor de la pantalla posicionado al principio de una nueva línea. Las convenciones de esta naturaleza facilitan la reutilización de software, un objetivo clave de los ambientes de desarrollo de software.

**2.4** Establezca sangrías en el cuerpo de cada función, un nivel hacia adentro de la llave que define el cuerpo de la función (nosotros recomendamos tres espacios). Esto hará que la estructura funcional de un programa resalte, y ayudará a que los programas sean más fáciles de leer.

**2.5** Establezca una convención para el tamaño de la sangría que usted prefiera, y aplique de manera uniforme dicha convención. Puede utilizar la tecla de tabulación para generar la sangría, pero los saltos de tabulación pueden variar. Nosotros le recomendamos que utilice saltos de tabulación de 1/4 de pulgada, o que cuente tres espacios para formar los niveles de las sangrías.

**2.6** Elegir nombres de variables que tengan significado le ayuda a escribir programas “autodocumentados”; es decir, necesitará menos comentarios.

**2.7** La primera letra de un identificador utilizado como un nombre de variable sencillo debe ser minúscula. Más adelante asignaremos un significado especial a los identificadores que comienzan con una letra mayúscula, y a los identificadores que utilizan todas sus letras en mayúsculas.

**2.8** Los nombres de variables con muchas palabras pueden ayudarle a escribir un programa más legible. Evite juntar palabras diferentes como **`comisionestotales`**; mejor utilice las palabras separadas por un guión bajo como en **`comisiones_totales`**, o, si desea juntar las palabras, comience cada una con letras mayúsculas como en **`ComisionesTotales`**. Este último estilo es preferible.

**2.9** Separe las declaraciones y las instrucciones ejecutables de una función mediante una línea en blanco, para resaltar donde terminan las declaraciones y donde comienzan las instrucciones ejecutables.

**2.10** Coloque un espacio después de cada coma (**`,`**), para hacer que los programas sean más legibles.

**2.11** Coloque espacios a cada lado de un operador binario. Esto hace que el operador resalte, y hace más claro el programa.

**2.12** Coloque sangrías en el cuerpo de una instrucción **`if`**.

**2.13** Coloque una línea en blanco antes y después de cada instrucción **`if`**, para mejorar la legibilidad del programa.

**2.14** Aunque está permitido, en un programa no debe haber más de una instrucción por línea.

**2.15** Una instrucción larga puede distribuirse en varias líneas. Si una instrucción debe separarse a lo largo de varias líneas, elija límites que tengan sentido (como después de una coma, en una lista separada por comas). Si una instrucción se divide en dos o más líneas, coloque sangrías en todas las líneas subsiguientes.

**2.16** Revise la tabla de precedencia de operadores, cuando escriba expresiones que contengan muchos operadores. Confirme que los operadores de la expresión se aplican en el orden correcto. Si no está seguro del orden de evaluación de una expresión compleja, utilice paréntesis para agrupar expresiones. Asegúrese de recordar que algunos de los operadores de C, como el de asignación (**`=`**), asocian de derecha a izquierda, y no de izquierda a derecha.

## *TIP DE PORTABILIDAD*

**2.1** Utilice identificadores de 31 caracteres o menos. Esto le ayudará a garantizar la portabilidad y puede evitar algunos problemas sutiles de programación.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**2.1** Complete los espacios en blanco.

a) Todo programa en C comienza su ejecución en la función ____________.
b) La ____________ comienza el cuerpo de toda función, y la ____________ finaliza el cuerpo de toda función.
c) Toda instrucción finaliza con un ____________.
d) La función ____________ de la biblioteca estándar despliega información en la pantalla.
e) La secuencia de escape **`\n`** representa una ____________, la cual ocasiona que el cursor se coloque al principio de la siguiente línea de la pantalla.
f) La función ____________ de la biblioteca estándar se utiliza para obtener datos desde el teclado.
g) El especificador de conversión ____________ se utiliza en una cadena de control de formato de **`scanf`** para indicar que se introducirá un entero, y en una cadena de control de formato de **`printf`** para indicar que el resultado será un entero.
h) Siempre que un nuevo valor se coloca en una posición de memoria, ese valor sobrescribe al anterior. Dicho proceso se conoce como lectura ____________.
i) Cuando un valor se lee desde una posición de memoria, el valor que se encuentra en esa posición se preserva; a esto se le llama lectura ____________.
j) La instrucción ____________ se utiliza para tomar decisiones.

**2.2** Diga si los siguientes enunciados son *verdaderos* o *falsos*. Si son *falsos*, explique por qué.

a) Cuando se llama a la función **`printf`**, ésta siempre comienza la impresión al principio de una nueva línea.
b) Cuando se ejecuta un programa, los comentarios ocasionan que la computadora imprima el texto encerrado entre **`/*`** y **`*/`** sobre la pantalla.
c) Cuando la secuencia de escape **`\n`** se utiliza en una cadena de control de formato **`printf`**, ésta ocasiona que el cursor se coloque al principio de la siguiente línea de la pantalla.
d) Todas las variables deben declararse, antes de que se utilicen.
e) A todas las variables se les debe asignar un tipo cuando se declaran.

f) C considera idénticas a las variables **numero** y **NuMEro**.
g) Las declaraciones pueden aparecer en cualquier parte del cuerpo de una función.
h) Todos los argumentos que se encuentran después de la cadena de control de formato en una función **printf** deben ser precedidos por un amperson (&).
i) El operador módulo (%) puede utilizarse sólo con operandos enteros.
j) Los operadores aritméticos *, /, %, + y – tienen el mismo nivel de precedencia.
k) Los siguientes nombres de variables son idénticos en todos los sistemas ANSI C.
**esteesunnombresuperduperlargo1234567**
**esteesunnombresuperduperlargo1234568**
l) Un programa que imprime tres líneas como resultado debe contener tres instrucciones **printf**.

**2.3** Escriba una sola instrucción de C para hacer lo que indican los siguientes enunciados:
a) Declare las variables **c**, **estaVariable**, **q76354** y **numero** como de tipo **int**.
b) Indique al usuario que introduzca un entero. Finalice su mensaje de indicaciones con dos puntos (**:**), seguidos por un espacio, y deje el cursor posicionado después del espacio.
c) Lea un entero introducido desde el teclado y almacene su valor en la variable entera **a**.
d) Si **numero** no es igual que **7**, imprima **"La variable numero no es igual que 7"**.
e) En una línea, imprima el mensaje **"Este es un programa en C"**.
f) En dos líneas, imprima el mensaje **"Este es un programa en C"**, de tal forma que la primera línea termine en "programa".
g) Imprima el mensaje **"Este es un programa en C"**, de tal forma que cada palabra aparezca en una línea diferente.
h) Imprima el mensaje **"Este es un programa en C"**, de tal forma que cada palabra aparezca separada por un salto del tabulador.

**2.4** Escriba una instrucción (o comentario) para realizar lo siguiente:
a) Indique que el programa calculará el producto de tres enteros.
b) Declare las variables **x**, **y**, **z** y **resultado** de tipo **int**.
c) Indique al usuario que introduzca tres enteros.
d) Lea tres enteros introducidos desde el teclado y almacénelos en las variables **x**, **y** y **z**.
e) Calcule el producto de los tres entero contenidos en las variables **x**, **y**, **z**, y asigne el resultado a la variable **resultado**.
f) Imprima **"El producto es"**, seguido del valor de la variable entera **resultado**.

**2.5** Escriba un programa completo que calcule el producto de tres enteros, utilizando las instrucciones que escribió en el ejercicio 2.4.

**2.6** Identifique y corrija los errores de cada una de las siguientes instrucciones:
a)
```
printf( "El valor es %d\n, &numero );
```
b)
```
scanf( "%d%d", &numero1, numero2 );
```
c)
```
if ( c < 7 );
    printf( "C es menor que 7\n" );
```
d)
```
if ( c => 7 );
    printf( "C es mayor o igual que 7\n" );
```

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**2.1** a) **main**. b) Llave izquierda (**{**), Llave derecha (**}**). c) Punto y coma. d) **printf**. e) Nueva línea. f) **scanf**. g) **%d**. h) Destructiva. i) No destructiva. j) **if**.

**2.2** a) Falso. La función **printf** siempre comienza a imprimir en donde se encuentra posicionado el cursor.
b) Falso. Los comentarios no ocasionan que se realice acción alguna cuando se ejecuta el programa.
c) Verdadero.
d) Verdadero.
e) Verdadero.
f) Falso. C es sensible a mayúsculas y minúsculas, por lo que estas variables son únicas.
g) Falso. Las declaraciones deben aparecer después de la llave izquierda que corresponde al cuerpo de la función, y antes de cualquier instrucción ejecutable.
h) Falso. Los argumentos de una función **printf**, en general no deben ser precedidos por un amperson. Los argumentos que siguen a la cadena de control de formato de una función **scanf**, por lo general deben ser precedidos por un amperson. Explicaremos algunas excepciones a estas reglas en los capítulos 6 y 7.

i) Verdadero.
j) Falso. Los operadores *, / y % tienen el mismo nivel de precedencia, y los operadores + y – tienen un nivel de precedencia más bajo.
k) Falso. Algunos sistemas pueden establecer diferencias entre identificadores mayores a 31 caracteres.
l) Falso. Una instrucción **`printf`** con múltiples secuencias de escape **`\n`**, puede imprimir varias líneas.

**2.3**
a) **`int c, estaVariable, q76354, numero;`**
b) **`printf( “Escriba un entero: “ );`**
c) **`scanf( “%d”, &a );`**
d) **`if( numero != 7 )`**
**`printf( “La variable numero no es igual que 7.\n”  );`**
e) **`printf( “Este es un programa en C.\n” );`**
f) **`printf( “Este es un programa\nen C.\n” );`**
g) **`printf( “Este\nes\nun\nprograma\nen\nC.\n” );`**
h) **`printf( “Este\tes\tun\tprograma\ten\tC.\n );`**

**2.4**
a) **`/* Calcula el producto de tres enteros */`**
b) **`int x, y, z, resultado;`**
c) **`printf( “Introduzca tres enteros: “ );`**
d) **`scanf( “%d%d%d”, &x, &y, &z );`**
e) **`resultado = x * y * z;`**
f) **`printf( “El producto es %d\n”, resultado );`**

**2.5** Ver abajo.

```
/* Calcula el producto de tres enteros */
#include <stdio.h>

int main( )
{
     int x, y, z, resultado; /* declara variables */

printf( “Introduzca tres enteros: “ ); /* indicador */
scanf( “%d%d%d”, &x, &y, &z ); /* lee tres enteros */
resultado = x * y * z; /* multiplica los valores */
printf( “El producto es %d\n”, resultado ); /* despliega el resultado */

return 0;
}
```

**2.6**
a) Error: **`&numero`**. Corrección: elimine el **`&`**. Más adelante explicaremos las excepciones a esto.
b) Error: **`numero2`** no tiene un amperson. Corrección: **`numero2`** debe aparecer como **`&numero2`**. Más adelante explicaremos las excepciones a esto.
c) Error: El punto y coma que se encuentra después del paréntesis derecho de la condición que se encuentra en la instrucción **`if`**. Corrección: elimine el punto y coma que se encuentra después del paréntesis derecho. [*Nota:* El resultado de este error es que la instrucción **`printf`** se ejecutará, independientemente de que la condición de la instrucción **`if`** sea verdadera. El punto y coma después del paréntesis se considera como una instrucción vacía; es decir, una instrucción que hace nada.]
d) Error: El operador de relación => debe cambiar a >= (mayor o igual que).

## *EJERCICIOS*

**2.7** Identifique y corrija los errores de cada uno de los siguientes ejercicios (*Nota:* Puede haber más de un error en cada ejercicio.)
a) **`scanf( “d”, valor );`**
b) **`printf( “El producto de %d y %d es %d\n, x, y );`**
c) **`primerNumero + segundoNumero = sumaDeNumeros`**
d) **`if ( numero => masGrande )`**
**`masGrande == numero;`**

e) **`*/ Programa para determinar el número más grande de tres enteros /*`**
f) **`Scanf( %d", unEntero );`**
g) **`printf( "El residuo de %d entre %d es\n", x, y, x % y );`**
h) **`if ( x = y );`**
**`printf( %d es igual que %d\n", x, y );`**
i) **`printf( "La suma es %d\n," x + y );`**
j) **`Printf( "El valor que escribió es: %d\n, &valor );`**

**2.8** Complete los espacios en blanco:
a) Los ____________ se utilizan para documentar un programa y para mejorar su legibilidad.
b) La función que se utiliza para desplegar información en la pantalla es ____________.
c) En C, una instrucción para tomar decisiones es ____________.
d) En general, las instrucciones ____________ son quienes realizan los cálculos.
e) La función ____________ introduce valores desde el teclado.

**2.9** Escriba una sola instrucción o línea de C que realice lo siguiente:
a) Imprima el mensaje **`"Escriba dos números"`**.
b) Asigne el producto de las variables **`b`** y **`c`** a la variable **`a`**.
c) Indique que un programa realiza un cálculo de nómina (es decir, utilice texto que ayude a documentar un programa).
d) Escriba tres valores enteros desde el teclado y coloque estos valores en las variables enteras **`a`**, **`b`** y **`c`**.

**2.10** Indique cuáles de las siguientes oraciones son *verdaderas* y cuáles son *falsas*. Si son *falsas*, explique su respuesta.
a) Los operadores de C se evalúan de izquierda a derecha.
b) Los siguientes son nombres de variables válidos: **`_guion_bajo_`**, **`m928134`**, **`t5`**, **`j7`**, **`sus_ventas`**, **`su_cuenta_total`**, **`a`**, **`b`**, **`c`**, **`z`**, **`z2`**.
c) La instrucción **`printf("a = 5;");`** es un típico ejemplo de una instrucción de asignación.
d) Una expresión aritmética válida que no contiene paréntesis se evalúa de izquierda a derecha.
e) Los siguientes son nombres no válidos de variables: **`3g`**, **`87`**, **`67h2`**, **`h22`**, **`2h`**.

**2.11** Complete los espacios en blanco:
a) ¿Qué operaciones aritméticas se encuentran en el mismo nivel de precedencia que la multiplicación? ____________.
b) En una expresión aritmética, cuando los paréntesis están anidados, ¿qué conjunto de paréntesis se evalúa primero?
c) Una posición en la memoria de la computadora que contiene diferentes valores en diferentes momentos, a lo largo de la ejecución de un programa se conoce como ____________.

**2.12** ¿Qué se imprime cuando se ejecuta cada una de las siguientes instrucciones? Si no se imprime algo, entonces responda "nada". Suponga que **`x = 2`** y **`y = 3`**.
a) **`printf( "%d", x );`**
b) **`printf( "%d", x + x );`**
c) **`printf( "x=" );`**
d) **`printf( "x=%d", x );`**
e) **`printf( "%d = %d", x + y, y + x );`**
f) **`z = x + y;`**
g) **`scanf( "%d%d", &x, &y );`**
h) **`/* printf( "x + y = %d", x + y ); */`**
i) **`printf( "\n");`**

**2.13** ¿Cuáles de las siguientes instrucciones de C contienen variables involucradas con la lectura destructiva?
a) **`scanf( "%d%d%d%d%d", &b, &c, &d, &e, &f );`**
b) **`p = i + j + k + 7;`**
c) **`printf( "Lectura destructiva" );`**
d) **`printf( "a = 5" );`**

**2.14** Dada la ecuación $y = ax^3+7$, ¿cuál de las siguientes son instrucciones correctas en C para esta ecuación?
a) **`y = a * x * x * x + 7;`**
b) **`y = a * x * x * ( x + 7 );`**
c) **`y = ( a * x ) * x * ( x + 7 );`**
d) **`y = ( a * x ) * x * x + 7;`**
e) **`y = a * ( x * x * x ) + 7;`**
f) **`y = a * x * ( x * x + 7 );`**

**2.15** Establezca el orden de evaluación de los operadores en cada una de las siguientes instrucciones de C, y muestre el valor de **`x`** después de que se realice cada instrucción.

a) **`x = 7 + 3 * 6 / 2 – 1;`**
b) **`x = 2 % 2 + 2 * 2 – 2 / 2;`**
c) **`x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`**

**2.16** Escriba un programa que pida al usuario escribir dos números, que obtenga los dos números por parte del usuario, y que imprime la suma, el producto, la diferencia, el cociente y el residuo de los dos números.

**2.17** Escriba un programa que imprima los números del 1 al 4 en la misma línea. Escriba el programa utilizando los siguientes métodos:

a) Mediante una instrucción **`printf`** sin especificadores de conversión.
b) Mediante una instrucción **`printf`** con cuatro especificadores de conversión.
c) Mediante cuatro instrucciones **`printf`**.

**2.18** Escriba un programa que pida al usuario que introduzca dos enteros, que obtenga los números por parte del usuario, después que imprima las palabras "**`es más grande`**". Si los números son iguales, que imprima el mensaje "**`Estos números son iguales`**". Solamente utilice la forma de selección simple de la instrucción **`if`**, que aprendió en este capítulo.

**2.19** Escriba un programa que introduzca tres diferentes enteros desde el teclado, después que imprima la suma, el promedio, el producto, el número más pequeño y el más grande de éstos. Solamente utilice la forma de selección simple de la instrucción **`if`**, que aprendió en este capítulo. El diálogo en la pantalla debe aparecer de la siguiente forma:

```
Escriba tres enteros diferentes: 13 27 14
La suma es 54
El promedio es 18
El producto es 4914
El número más pequeño es 13
El número más grande es 27
```

**2.20** Escriba un programa que lea el radio de un círculo y que imprima el diámetro, la circunferencia y el área de ese círculo. Utilice el valor constante de 3.14159 para $\pi$. Realice cada uno de estos cálculos dentro de instrucción(es) **`printf`**, y utilice el especificador de conversión **`%f`**. [*Nota:* En este capítulo sólo explicamos constantes y variables enteras. En el capítulo 3 explicaremos los números de punto flotante, es decir, valores que pueden tener puntos decimales.]

**2.21** Escriba un programa que imprima una caja, un óvalo, una flecha y un diamante como los siguientes:

```
*********          ***              *              *
*       *        *     *           ***            * *
*       *       *       *         *****          *   *
*       *       *       *           *           *     *
*       *       *       *           *          *       *
*       *       *       *           *           *     *
*       *       *       *           *            *   *
*       *        *     *            *             * *
*********          ***              *              *
```

**2.22** ¿Qué imprime el siguiente código?

**`printf( "*\n**\n***\n****\n*****\n" );`**

**2.23** Escriba un programa que lea cinco enteros y que después imprima el número más grande y el más pequeño del grupo. Utilice sólo técnicas de programación que haya aprendido en este capítulo.

**2.24** Escriba un programa que lea un entero y que determine e imprima si es par o impar. [*Pista:* Utilice el operador módulo. Un número par es un múltiplo de dos. Cualquier múltiplo de 2 arroja un residuo de cero, cuando se divide entre 2.]

**2.25** Imprima sus iniciales en mayúsculas de imprenta, de manera que apunten hacia la parte inferior de la páginas (acostadas). Construya cada mayúscula de imprenta con la letra que ésta representa, de la siguiente forma:

```
PPPPPPPPP
    P   P
    P   P
    P   P
     P P

  JJ
 J
J
 J
  JJJJJJJ

DDDDDDDDD
D       D
D       D
 D     D
  DDDDD
```

**2.26** Escriba un programa que lea dos enteros y que determine e imprima si el primero es múltiplo del segundo. [*Pista:* Utilice el operador módulo.]

**2.27** Despliegue el siguiente patrón de diseño mediante ocho instrucciones **`printf`**, y después despliegue el mismo patrón con el menor número posible de instrucciones **`printf`**.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

**2.28** Distinga entre los términos error fatal y no fatal. ¿Por qué podría usted preferir experimentar un error fatal a un no fatal?

**2.29** He aquí un avance. En este capítulo aprendió acerca de enteros y del tipo **`int`**. C también puede representar letras mayúsculas, minúsculas, y una considerable variedad de símbolos especiales. C utiliza internamente enteros pequeños para representar cada carácter. Al conjunto de caracteres que utiliza una computadora y a las representaciones enteras para esos caracteres se les conoce como conjunto de caracteres de la computadora. Por ejemplo, usted puede imprimir el entero equivalente a la A mayúscula, si ejecuta la instrucción:

```
printf( “%d”, ‘A’ );
```

Escriba un programa en C que imprima los enteros equivalentes a algunas letras mayúsculas, minúsculas, dígitos y símbolos especiales. Como mínimo, determine los enteros equivalentes de las siguientes: **`A B C a b c 0 1 2 $ * + /`** y el carácter espacio en blanco.

**2.30** Escriba un programa que introduzca un número de cinco dígitos, que separe el número en sus dígitos individuales y que despliegue los dígitos separados entre sí mediante tres espacios cada uno. [*Pista:* Utilice combinaciones de la división entera y el operador módulo.] Por ejemplo, si el usuario escribe **`42139`**, el programa debe imprimir

```
4   2   1   3   9
```

**2.31** Utilice sólo las técnicas que aprendió en este capítulo para escribir un programa que calcule los cuadrados y los cubos de los números 0 a 10, y que utilice tabuladores para desplegar la siguiente tabla de valores:

```
numero   cuadrado    cubo
0        0           0
1        1           1
2        4           8
3        9           27
4        16          64
5        25          125
6        36          216
7        49          343
8        64          512
9        81          729
10       100         1000
```



# Desarrollo de programas estructurados en C

## Objetivos

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos a través del proceso de mejoramiento arriba abajo, paso a paso.
- Utilizar las instrucciones de selección **`if`** e **`if…else`** para seleccionar acciones.
- Utilizar la instrucción de repetición **`while`** para ejecutar repetidamente las instrucciones de un programa.
- Comprender la repetición controlada por contador y la repetición controlada por centinela.
- Comprender la programación estructurada.
- Utilizar los operadores de incremento, decremento y asignación.




*El secreto del éxito es la constancia.*
Benjamin Disraeli

*Movámonos un lugar hacia delante.*
Lewis Carroll

*La rueda ha completado el círculo.*
William Shakespeare
*El rey Lear*

*¿Cuántas manzanas cayeron en la cabeza de Newton antes de que tuviera la idea?*
Robert Frost
(Comentario)

## Plan general



## 3.1 Introducción

Antes de escribir un programa para resolver un problema en particular, es esencial que comprendamos el problema y el método para resolver dicho problema. Los dos capítulos siguientes explican las técnicas que facilitan el desarrollo de programas estructurados de computadora. En la sección 4.12, presentamos un resumen sobre programación estructurada, el cual une las técnicas que desarrollamos en éste y en el capítulo 4.

## 3.2 Algoritmos

La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones en un orden específico. Al *procedimiento* para resolver un problema en términos de:

1. Las *acciones* a ejecutar.
2. El *orden* en el cual se llevan a cabo dichas acciones.

se le llama *algoritmo*. El siguiente ejemplo demuestra que es importante especificar correctamente el orden en el que se deben ejecutar las acciones.

Considere el algoritmo "levantarse y arreglarse" que sigue un joven ejecutivo para salir de la cama e ir a su trabajo:

*Levantarse de la cama.*
*Quitarse la pijama.*
*Bañarse.*
*Vestirse.*
*Desayunar.*
*Manejar hacia el trabajo.*

Esta rutina hace que el ejecutivo vaya al trabajo bien preparado para tomar decisiones críticas. Sin embargo, suponga que sigue los mismos pasos en un orden ligeramente diferente:

*Levantarse de la cama.*
*Quitarse la pijama.*
*Vestirse.*
*Bañarse.*
*Desayunar.*
*Manejar hacia el trabajo.*

En este caso, nuestro joven ejecutivo llega al trabajo empapado. A la especificación del orden en el cual se ejecutan las instrucciones dentro de un programa de computadora se le llama *control del programa*. En este capítulo y en el siguiente, investigaremos las capacidades de control del programa de C.

## 3.3 Pseudocódigo

El *pseudocódigo* es un lenguaje artificial e informal que ayuda a los programadores a desarrollar algoritmos. El pseudocódigo es similar al inglés común; es conveniente y sencillo, aunque no es un lenguaje de programación real.

Los programas en pseudocódigo no se ejecutan en las computadoras, sino que sólo ayudan al programador a “resolver” un programa antes de intentar escribirlo en un lenguaje de programación como C. En este capítulo, proporcionamos muchos ejemplos respecto a la manera efectiva de utilizar el pseudocódigo para desarrollar programas estructurados en C.

El pseudocódigo sólo consiste en caracteres, de manera que los programadores pueden introducir los programas en pseudocódigo a la computadora mediante un programa de edición. La computadora puede desplegar o imprimir una copia reciente del pseudocódigo cuando sea necesario. Un programa en pseudocódigo cuidadosamente preparado puede convertirse fácilmente en su correspondiente programa en C. En muchos casos esto se hace mediante un simple reemplazo de las instrucciones en pseudocódigo por sus equivalentes en C.

El pseudocódigo sólo consiste en instrucciones de acción, es decir, aquellas que se ejecutan cuando el programa se convirtió de pseudocódigo a C y se ejecutan en C. Las declaraciones no son instrucciones ejecutables. Son mensajes para el compilador. Por ejemplo, la definición

```
int i;
```

simplemente le indica al compilador el tipo de la variable **i**, e instruye al compilador para que reserve el espacio en memoria para la variable. Sin embargo, esta definición no provoca la ejecución de acción alguna (tal como una entrada, salida, o cálculo) cuando se ejecuta el programa. Algunos programadores eligen mostrar cada variable y mencionar de manera breve el propósito de cada una al principio del pseudocódigo del programa. De nuevo, el pseudocódigo es una ayuda para el desarrollo de programas.

## 3.4 Estructuras de control

Por lo general, las instrucciones dentro de un programa se ejecutan una a una en el orden en que están escritas. A esto se le llama *ejecución secuencial*. Varias instrucciones de C, que explicaremos más adelante, permiten al programador especificar que la siguiente instrucción a ejecutarse debe ser otra y no la siguiente en la secuencia. A esto se le llama *transferencia de control*.

Durante la década de los sesentas, se hizo claro que el uso indiscriminado de transferencias de control era el origen de un gran número de dificultades que experimentaban los grupos de desarrollo de software. El dedo de la culpa apunto hacia la *instrucción* ***goto***, que permite al programador especificar una transferencia de control a un amplio margen de destinos posibles dentro de un programa. La idea de la *programación estructurada* se convirtió casi en un sinónimo de la “*eliminación del* ***goto***”.

Las investigaciones de Bohm y Jacopini[1] demostraron que los programas se pueden escribir sin instrucción **goto** alguna. El reto para los programadores de la época era modificar sus estilos hacia una “programación con menos instrucciones **goto**”. No fue sino hasta la década de los setenta que los profesionales de la progra-

1. Bohm, C., y G. Jacopini, “Flow diagrams, Turing Machines, and Languages with Only Two Formation Rules”, *Communications of the ACM*, Vol. 9, No. 5, mayo de 1996, pp. 336 a 371.

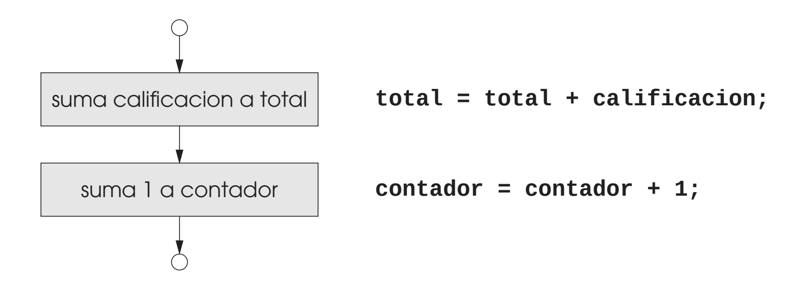



**Figura 3.1** Diagrama de flujo de la estructura secuencial de C.

mación comenzaron a tomar en serio a la programación estructurada. Los resultados fueron impresionantes, los grupos de desarrollo de software reportaron una reducción en los tiempos de desarrollo, la entrega más oportuna de los sistemas y el apego más frecuente al presupuesto de los proyectos de software. La clave de este éxito fue simplemente que los programas producidos mediante técnicas estructuradas eran más claros, más fáciles de mantener y depurar, y tenían más probabilidades de estar libres de errores desde el principio.

El trabajo de Bohm y Jacopini demostró que todos los programas se podían escribir en términos de sólo tres *estructuras de control*, a saber, la *estructura secuencial*, la *estructura de selección*, y la *estructura de repetición*. La estructura de secuencia se encuentra esencialmente dentro de C. A menos que se le indique lo contrario, la computadora ejecuta de manera automática las instrucciones en C, una a una, en el orden en que están escritas. El segmento de *diagrama de flujo* de la figura 3.1 muestra la estructura secuencial de C.

Un diagrama de flujo es una representación gráfica de un algoritmo o de una porción de un algoritmo. Los diagramas de flujo se dibujan mediante símbolos de propósito especial tales como rectángulos, rombos, óvalos, y pequeños círculos; estos símbolos se conectan mediante flechas llamadas *líneas de flujo*.

Como el pseudocódigo, los diagramas de flujo son útiles para desarrollar y representar algoritmos, aunque la mayoría de los programadores prefieren el pseudocódigo. Los diagramas de flujo muestran claramente la manera en que operan las estructuras de control; esto es lo único para lo que los utilizaremos en este libro.

Considere el diagrama de flujo para la estructura secuencial de la figura 3.1. Utilizamos el *símbolo rectángulo*, también llamado *símbolo de acción*, para indicar cualquier tipo de acción, incluyendo una operación de cálculo o de entrada/salida. Las líneas de flujo de la figura indican el orden en el que se realizan las acciones (primero, se suma **`calificacion`** a **`total`** y posteriormente se suma **`1`** a **`contador`**. C nos permite tener en una estructura secuencial tantas acciones como deseemos. Como veremos más adelante, en cualquier lugar en donde coloquemos una acción, también podemos colocar muchas acciones en secuencia.

Cuando dibujamos un diagrama de flujo que representa un algoritmo completo, el primer símbolo que se utiliza es un *óvalo* que contiene la palabra "Inicio"; y el último símbolo que se utiliza es un *óvalo* que contiene la palabra "Fin". Cuando dibujamos sólo una porción de un algoritmo, como en la figura 3.1, se omiten los símbolos de óvalo y se emplean pequeños *círculos* también llamados *símbolos conectores*.

Quizá el símbolo más importante dentro de un diagrama de flujo es el *rombo*, también llamado *símbolo de decisión*, el cual indica que se va tomar una decisión. Explicaremos el símbolo de decisión en la siguiente sección.

C proporciona tres tipos de estructuras de selección en forma de instrucciones. La instrucción de selección **`if`** (sección 3.5) realiza (selecciona) una acción si la condición es verdadera, o ignora la acción si la condición es falsa. La instrucción de selección **`if…else`** (sección 3.6) realiza una acción si la condición es verdadera y realiza una acción diferente si la condición es falsa. La instrucción de selección **`switch`** (la cual explicaremos en el capítulo 4) realiza una de muchas acciones dependiendo del valor de una expresión. A la instrucción **`if`** se le conoce como una *instrucción de selección simple*, debido a que selecciona o ignora una sola acción. A la instrucción **`if…else`** se le conoce como una *instrucción de selección doble*, debido a que selecciona entre dos acciones diferentes. A la instrucción **`switch`** se le conoce como una *instrucción de selección múltiple*, debido a que selecciona entre muchas acciones diferentes.

C proporciona tres tipos e estructuras de repetición en forma de instrucciones, a saber, **`while`** (sección 3.7), **`do…while`**, y **`for`** (estas dos últimas las explicaremos en el capítulo 4).

Y esto es todo. C sólo tiene siete instrucciones de control: Secuencia, tres tipos de selección y tres tipos de repetición. Cada programa en C está formado por la combinación de tantas instrucciones de control como sea adecuado para el algoritmo que implementa el programa. Así como en la estructura secuencial de la figura 3.1, veremos que la representación en un diagrama de flujo de cada una de las instrucciones de control tiene dos círculos pequeños, uno en el punto de entrada de la instrucción de control y otro en el punto de salida. Estas *instrucciones de control de entrada simple/salida simple* hacen fácil la construcción de programas. Los segmentos de diagramas de flujo correspondientes a instrucciones de control se pueden unir unos con otros, conectando el punto de salida de una instrucción de control con el punto de entrada de la siguiente. Esto se parece mucho a la manera en la que un niño apila bloques de construcción, de manera que a esto le llamamos *apilamiento de estructuras de control*. Aprenderemos que solamente existe otra manera de conectar instrucciones de control, esto es, mediante un método llamado *anidamiento de instrucciones de control*. Así, cualquier programa en C que necesitemos desarrollar se puede construir a partir de sólo siete tipos diferentes de instrucciones de control combinadas de dos maneras posibles. Ésta es la esencia de la simplicidad.

## 3.5 La instrucción de selección `if`

Las estructuras de selección se utilizan para elegir entre diversos cursos de acción. Por ejemplo, suponga que la calificación mínima para aprobar un examen es 60. La instrucción en pseudocódigo es

> ***if*** *calificación del estudiante es mayor o igual que 60*
>     *imprime "Aprobado"*

y determina si la condición "calificación del estudiante es mayor o igual que 60" es verdadera o falsa. Si la condición es verdadera, entonces se imprime "Aprobado", y se "ejecuta" la siguiente instrucción en pseudocódigo (recuerde que el pseudocódigo no es un lenguaje de computadora real). Si la condición es falsa, se ignora la impresión y se ejecuta la siguiente instrucción en pseudocódigo. Observe que la segunda línea de esta estructura de selección tiene sangría. Tal sangrado es opcional, pero es muy recomendable ya que ayuda a enfatizar la estructura interna de los programas estructurados. Aplicaremos convenciones de sangrado de manera cuidadosa a lo largo del libro. El compilador de C ignora los *caracteres blancos* como los espacios en blanco, tabuladores y nuevas líneas utilizadas para el sangrado y la distribución vertical.



### Buena práctica de programación 3.1

*La aplicación consistente de convenciones para el sangrado, mejora de manera importante la claridad del programa. Le sugerimos un tabulador de tamaño fijo de 1/4 de pulgada o tres espacios en blanco por sangrado. En este libro, utilizamos tres espacios en blanco por sangrado.*

La instrucción **`if`** del pseudocódigo anterior se puede escribir en C de la siguiente manera:

```
if ( calificacion >= 60 )
   printf( "Aprobado\n" );
```

Observe que el código en C se parece mucho al pseudocódigo. Ésta es una de las propiedades del pseudocódigo que lo hacen una herramienta de desarrollo tan útil.



### Buena práctica de programación 3.2

*A menudo, el pseudocódigo se utiliza para "plantear" un programa durante el proceso de diseño. Posteriormente el programa en pseudocódigo se convierte a C.*

El diagrama de flujo de la figura 3.2 muestra la instrucción de selección simple **`if`**. Este diagrama de flujo contiene lo que quizá es el símbolo más importante de los diagramas de flujo, el *rombo*, también llamado *símbolo de decisión*, el cual indica que se va a tomar una decisión. El símbolo de decisión contiene una expresión, tal como una condición, que indica la decisión que se debe tomar. El símbolo de decisión contiene dos líneas de flujo que emergen de él. Uno indica la dirección que se debe tomar cuando la expresión dentro del símbolo es verdadera; la otra indica la dirección que se debe tomar cuando la expresión es falsa. En el capítulo 2 aprendimos que las decisiones se pueden basar en condiciones que contienen operadores de relación o de igualdad. De hecho, una decisión se puede basar en cualquier expresión; si la expresión es igual a cero, se trata como falsa, y si la expresión es diferente de cero, se trata como verdadera.

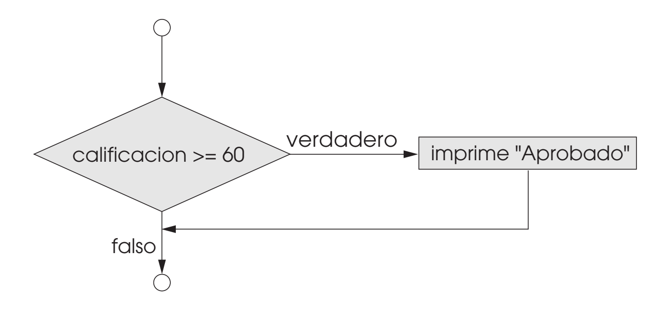



**Figura 3.2** Diagrama de flujo de la instrucción de selección simple **`if`**.

Observe que la instrucción **`if`** también es una estructura de entrada simple/salida simple. Pronto aprenderemos que los diagramas de flujo para las estructuras de control restantes también pueden contener (además de pequeños círculos y líneas de flujo) solamente rectángulos para indicar las acciones que se deben realizar, y rombos para indicar las decisiones que se deben tomar. Éste es el modelo de programación acción/decisión que hemos estado enfatizando.

Podemos visualizar siete contenedores, cada uno con diagramas de flujo de uno de los siete tipos e instrucciones de control. Estos segmentos de diagramas de flujo están vacíos, nada está escrito dentro de los rectángulos ni dentro de los rombos. Entonces, la tarea del programador es la de ensamblar un programa, partiendo de tantas instrucciones de control de cada tipo como lo requiera el algoritmo, combinar dichas instrucciones de control de sólo dos maneras posibles (apilado o anidado), y entonces llenar las acciones y las decisiones de manera apropiada para el algoritmo. Explicaremos la variedad de formas en las cuales podemos escribir las acciones y las decisiones.

## 3.6 La instrucción de selección `if…else`

La instrucción de selección **`if`** realiza una acción indicada, sólo cuando la condición es verdadera; de lo contrario, se ignora dicha acción. La instrucción de selección **`if…else`** permite al programador especificar que se realizarán acciones diferentes cuando la condición sea verdadera y cuando la condición sea falsa. Por ejemplo, la instrucción en pseudocódigo

*if calificación del estudiante es mayor o igual que 60*
    *Imprime "Aprobado"*
*else*
    *Imprime "Reprobado"*

imprime *Aprobado* si la calificación del estudiante es mayor o igual que 60, e imprime *Reprobado* si la calificación del estudiante es menor que 60. En cualquiera de los casos, después de que ocurre la impresión, se *ejecuta* la siguiente instrucción del pseudocódigo. Observe que también el cuerpo del *else* está sangrado. Independientemente de la convención de sangrado que utilice, debe utilizarla con cuidado a lo largo de sus programas. Es difícil leer un programa que no obedece reglas uniformes de espaciado.



**Buena práctica de programación 3.3**

*Coloque sangrías en las dos instrucciones que componen el cuerpo de una instrucción **`if…else`**.*



**Buena práctica de programación 3.4**

*Si existen muchos niveles de sangrado, cada nivel debe estar sangrado con el mismo número de espacios.*

La instrucción ***`if…else`*** del pseudocódigo anterior se puede escribir en C como:

```
if ( calificación >= 60 )
   printf( "Aprobado\n" );
else
   printf( "Reprobado\n" );
```

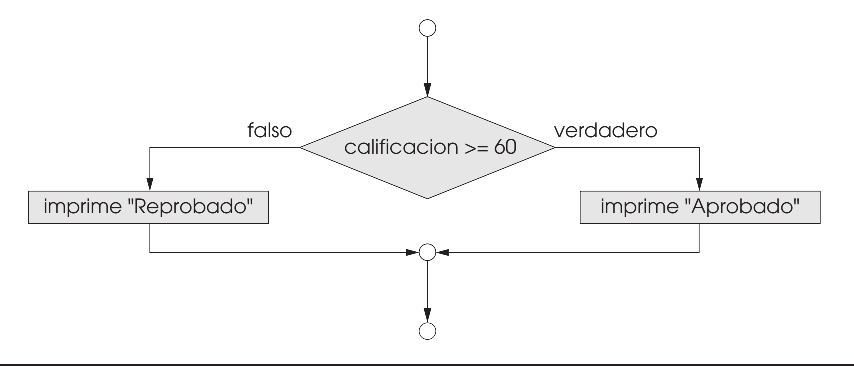



**Figura 3.3** Diagrama de flujo de la instrucción de selección doble **`if…else`**.

El diagrama de flujo de la figura 3.3 ilustra de manera clara el flujo de control de la instrucción **`if…else`**. Una vez más, observe que (además de los pequeños círculos y las flechas) los únicos símbolos en el diagrama de flujo son rectángulos (para las acciones) y un rombo (para la decisión). Continuaremos haciendo énfasis en este modelo de cómputo acción/decisión. De nuevo, imagine un contenedor profundo con tantas instrucciones de selección doble (representadas por segmentos de diagramas de flujo) como fueran necesarias para construir cualquier programa en C. Otra vez, el trabajo del programador es ensamblar estas instrucciones de selección (apilando y anidando) con otras instrucciones de control requeridas por el algoritmo, y llenar los rectángulos y los rombos vacíos con acciones y decisiones apropiadas para el algoritmo que va a implementar.

C proporciona el *operador condicional (**`?:`**)*, el cual está íntimamente relacionado con la instrucción **`if…else`**. El operador condicional es el único *operador ternario* de C, es decir, requiere tres operandos. Los operandos junto con el operador condicional forman una *expresión condicional*. El primer operando es una condición. El segundo operando es el valor para toda la expresión condicional, si la expresión es verdadera, y el tercer operando es el valor para toda la expresión condicional, si la condición es falsa. Por ejemplo, la instrucción **`printf`**

```
printf( “%s\n”, calificacion >= 60 ? “Aprobado” : “Reprobado” );
```

contiene una expresión condicional que evalúa la cadena literal “Aprobado”, si la condición **`calificacion >= 60`** es verdadera, y evalúa la cadena literal “Reprobado”, si la condición es falsa. La cadena de control de formato de **`printf`** contiene la especificación de conversión **`%s`** para imprimir los caracteres de la cadena. Por lo tanto, la instrucción **`printf`** anterior se ejecuta esencialmente de la misma forma que la instrucción **`if…else`**.

Los valores de una expresión condicional también pueden ser acciones a ejecutar. Por ejemplo, la expresión condicional

```
calificacion >= 60 ? printf( “Aprobado\n” ) : printf( “Reprobado\n” );
```

se lee “Si la calificación es mayor o igual que 60, entonces **`printf(“Aprobado\n”`**), de lo contrario **`printf(“Reprobado\n”)`**”. También esto se puede comparar con la instrucción **`if…else`** anterior. Veremos que los operadores condicionales pueden utilizarse en algunas situaciones en donde los **`if…else`** no.

Las *instrucciones* **`if…else`** *anidadas* evalúan múltiples casos al colocar instrucciones **`if…else`** dentro de otras instrucciones **`if…else`**. Por ejemplo, la instrucción siguiente en pseudocódigo imprime una **`A`** para las calificaciones mayores o iguales que **`90`**, **`B`** para las calificaciones mayores o iguales que **`80`**, **`C`** para las calificaciones mayores o iguales que **`70`**, **`D`** para las calificaciones mayores o iguales que **`60`**, y **`F`** para todas las demás calificaciones.

*if calificación del estudiante es mayor o igual que 90*
    *Imprime “A”*
*else*
    *if calificación del estudiante es mayor o igual que 80*
        *Imprime “B”*

*else*
*if calificación del estudiante es mayor o igual que 70*
*Imprime "C"*
*else*
*if calificación del estudiante es mayor o igual que 60*
*Imprime "D"*
*else*
*Imprime "F"*

Este pseudocódigo se puede escribir en C como:

```
if ( calificacion >= 90 )
   printf( "A\n" );
else
   if ( calificacion >= 80 )
      printf( "B\n" );
   else
      if ( calificacion >= 70 )
         printf( "C\n" );
      else
         if ( calificacion >= 60 )
            printf( "D\n" );
      else
            printf( "F\n" );
```

Si la variable **calificacion** es mayor o igual que **90**, las primeras cuatro condiciones serán verdaderas, pero sólo se ejecutará la instrucción **printf** después de la primera condición. Después de la ejecución del **printf** se ignora la parte **else** del **if…else** "externo". Muchos programadores en C prefieren escribir la instrucción **if** anterior como

```
if ( calificacion >= 90 )
   printf( "A\n" );
else if ( calificacion >= 80 )
   printf( "B\n" );
else if ( calificacion >= 70 )
   printf( "C\n" );
else if ( calificacion >= 60 )
   printf( "D\n" );
else
   printf( "F\n" );
```

En lo que respecta al compilador de C, ambas formas son equivalentes. La última forma es popular debido a que evita un sangrado profundo de código hacia la derecha. Dicho sangrado a menudo deja poco espacio en la línea, lo que provoca que las líneas se dividan y provoquen una menor claridad del programa.

La instrucción de selección **if** permite sólo una instrucción dentro del cuerpo. Para incluir varias instrucciones dentro del cuerpo de un **if**, encierre las instrucciones dentro de llaves (**{** y **}**). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama *instrucción compuesta* o *bloque*.

### Observación de ingeniería de software 3.1

*Una instrucción compuesta puede colocarse en cualquier parte de un programa en donde pueda colocarse una instrucción sencilla.*

El ejemplo siguiente incluye una instrucción compuesta en la parte **else** de una instrucción **if…else**.

```
if ( calificacion >= 60 )
   printf( "Aprobado.\n" );
else {
   printf( "Reprobado.\n" );
   printf( "Usted deberá tomar nuevamente el curso.\n" );
}
```

En este caso, si **calificacion** es menor que **60**, el programa ejecuta las dos instrucciones **printf** en el cuerpo del **else** e imprime

```
Reprobado.
Usted deberá tomar nuevamente el curso.
```

Observe las llaves que envuelven a las dos instrucciones de la cláusula *else*. Estas llaves son importantes. Sin las llaves, la instrucción

```
printf( "Usted deberá tomar nuevamente el curso.\n" );
```

estaría afuera del cuerpo de la parte **else** del **if**, y se ejecutaría sin importar si la calificación fuera o no menor que 60.



**Error común de programación 3.1**

*Olvidar una o las dos llaves que delimitan una instrucción compuesta.*

Un error de sintaxis se detecta mediante el compilador. Un error de lógica tiene efecto en tiempo de ejecución. Un error fatal de lógica provoca que el programa falle y termine de manera prematura. Un error no fatal de lógica permite al programa continuar la ejecución, pero produce resultados incorrectos.



**Error común de programación 3.2**

*Colocar un punto y coma después de la condición de una instrucción* **if** *provoca un error de lógica dentro de las instrucciones* **if** *de selección simple y un error de sintaxis en las instrucciones* **if** *de selección doble.*



**Tip para prevenir errores 3.1**

*Escribir las llaves inicial y final de instrucciones compuestas, antes de escribir las instrucciones individuales que van dentro de ellas, ayuda a evitar la omisión de una o ambas llaves, a prevenir errores de sintaxis y a prevenir errores de lógica (en donde se requieren ambas llaves).*



**Observación de ingeniería de software 3.2**

*Tal como una instrucción compuesta puede colocarse en cualquier parte en donde puede colocarse una instrucción sencilla, también es posible no tener instrucción alguna, es decir, tener una instrucción vacía. La instrucción vacía se representa colocando un punto y coma (***;***) en donde por lo general va la instrucción.*

## 3.7 La instrucción de repetición while

Una *instrucción de repetición* permite al programador especificar que una acción se va a repetir mientras una condición sea verdadera. La instrucción en pseudocódigo

*While existan más elementos en mi lista de compras*
    *Compra el siguiente elemento y márcalo en mi lista*

describe la repetición que ocurre durante un proceso de compras. La condición "existan más elementos en mi lista de compras" puede ser falsa o verdadera. Si es verdadera, entonces se realiza la acción "Compra el siguiente elemento y márcalo en mi lista". Esta acción se llevará a cabo de manera repetida mientras la condición sea verdadera. La(s) instrucción(es) contenida(s) dentro de la instrucción de repetición *while* constituyen el cuerpo de la instrucción. El cuerpo de la instrucción *while* puede ser una sola instrucción o una instrucción compuesta.

En algún momento, la condición será falsa (cuando el último elemento se compre y se marque en la lista). En este punto, termina la repetición, y se ejecuta la siguiente instrucción en pseudocódigo después de la estructura de repetición.



**Error común de programación 3.3**

*No proporcionar una acción dentro del cuerpo de una instrucción* **while** *que permita que ésta se haga falsa, ocasionará que dicha estructura de repetición no termine nunca; a esto se le conoce como "ciclo infinito".*



**Error común de programación 3.4**

*Escribir la palabra reservada* **while** *con una letra mayúscula, como en* **While** *(recuerde que C es un lenguaje sensible a mayúsculas y minúsculas). Todas las palabras reservadas de C tales como* **while**, **if** *y* **else** *contienen sólo letras minúsculas.*

Como ejemplo de un **while** real, considere un segmento de programa diseñado para encontrar la primera potencia de 2 que sea mayor que 1000. Suponga que la variable entera **producto** se inicializa en 2. Cuando finaliza la ejecución de la siguiente instrucción de repetición **while**, **producto** contendrá la respuesta deseada:

```
producto = 2;

while ( producto <= 1000 )
   producto = 2 * producto;
```

El diagrama de flujo de la figura 3.4 muestra de manera clara el flujo de control de la instrucción de repetición **while**. Una vez más, observe que (además de los pequeños círculos y las flechas) el diagrama de flujo contiene solamente un rectángulo y un rombo. El diagrama de flujo muestra de manera clara la repetición. La línea de flujo que surge del rectángulo se dirige hacia atrás; hacia la decisión que se evalúa una y otra vez en el ciclo, hasta que la decisión se hace falsa. En este punto, se abandona la instrucción **while** y se pasa el control a la siguiente instrucción del programa.

Al entrar en la instrucción **while** por primera vez, el valor de **producto** es 2. La variable **producto** se multiplica de manera repetida por 2, tomando los valores 4, 8, 16, 32, 64, 128, 256, 512 y 1024 de manera exitosa. Cuando **producto** toma el valor 1024, la condición **producto <= 1000** de la instrucción de repetición **while** se torna falsa. Esto termina la repetición, y el valor final de **producto** es 1024. La ejecución del programa continúa con la siguiente instrucción después del **while**.

## 3.8 Formulación de algoritmos: Ejemplo práctico 1 (repetición controlada por contador)

Para mostrar cómo se desarrollan los algoritmos, resolveremos distintas variantes del problema del promedio de calificaciones de una clase. Considere el siguiente enunciado del problema:

> *Un grupo de diez estudiantes realizó un examen. Usted tiene a su disposición las calificaciones (enteros en el rango de 0 a 100) de este examen. Determine el promedio de las calificaciones del grupo en este examen.*

El promedio del grupo es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe introducir cada una de las calificaciones, realizar el cálculo del promedio e imprimir el resultado.

Utilicemos pseudocódigo, listemos las acciones que vamos a llevar a cabo, y especifiquemos el orden en el que se deben ejecutar dichas acciones. Utilizamos el término *repetición controlada por contador* para introducir las calificaciones, una a la vez. Esta técnica utiliza una variable llamada *contador* para especificar el número de veces que se ejecuta un conjunto de instrucciones. En este ejemplo, la repetición termina cuando el contador, excede de 10. En esta sección simplemente presentamos el algoritmo en pseudocódigo (figura 3.5) y su correspondiente programa en C (figura 3.6). En la siguiente sección, mostramos cómo se desarrollaron los algoritmos. A menudo, a la repetición controlada por contador se le conoce como *repetición definida* debido a que se conoce el número de repeticiones antes de la ejecución del ciclo.

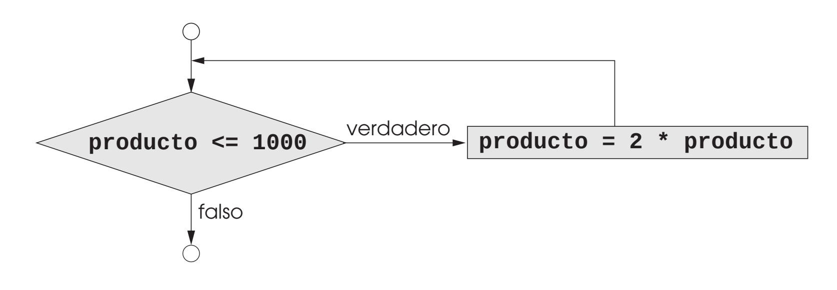



**Figura 3.4** Diagrama de flujo de la instrucción de repetición **while**.

*Establece total en cero*
*Establece contador de calificaciones en uno*

*While contador de calificaciones sea menor o igual que diez*
    *Introduce la siguiente calificación*
    *Suma la calificación a total*
    *Suma uno a contador de calificaciones*

*Establece el promedio del grupo dividido entre diez en total*
*Imprime el promedio del grupo*

**Figura 3.5** Algoritmo en pseudocódigo que utiliza una repetición controlada por contador para resolver el problema del promedio de calificaciones de un grupo.

```
/* Figura 3.6: fig03_06.c
   Programa para obtener el promedio de calificaciones de un grupo mediante
   una repetición controlada por contador */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   int contador;       /* número de la calificación siguiente */
   int calificacion;   /* valor de la calificación */
   int total;          /* suma de las calificaciones introducidas
                          por el usuario */
   int promedio;       /* promedio de las calificaciones */

   /* fase de inicialización */
   total    = 0; /* inicializa total */
   contador = 1; /* inicializa el contador del ciclo */

   /* fase de procesamiento */
   while ( contador <= 10 ) {       /* repite 10 veces */
      printf( "Introduzca la calificacion: " ); /* indicador para la
      entrada */
      scanf( "%d", &calificacion ); /* lee la calificación del usuario */
      total = total + calificacion; /* suma la calificación al total */
      contador = contador + 1;      /* incrementa el contador */
   } /* fin de while */

   /* fase de terminación */
   promedio = total / 10; /* división entera */

   printf( "El promedio del grupo es %d\n", promedio ); /* despliega el
   resultado */

   return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

**Figura 3.6** Programa en C y ejemplo de la ejecución para el problema del promedio de la clase mediante un contador controlado por repetición. (Parte 1 de 2.)

```
Introduzca la calificacion: 98
Introduzca la calificacion: 76
Introduzca la calificacion: 71
Introduzca la calificacion: 87
Introduzca la calificacion: 83
Introduzca la calificacion: 90
Introduzca la calificacion: 57
Introduzca la calificacion: 79
Introduzca la calificacion: 82
Introduzca la calificacion: 94
El promedio del grupo es 81
```

**Figura 3.6** Programa en C y ejemplo de la ejecución para el problema del promedio del grupo mediante una repetición controlada por contador. (Parte 2 de 2.)

Observe en el algoritmo, las referencias a un total y a un contador. Un *total* es una variable que se utiliza para acumular la suma de una serie de valores. Un contador es una variable que se utiliza para contar, en este caso, para contar el número de calificaciones introducidas. Por lo general, las variables que se utilizan para almacenar totales se deben inicializar en cero antes de emplearlas dentro del programa; de lo contrario, la suma incluirá el valor previo almacenado en la dirección de memoria reservada para el total. Por lo general, las variables contadoras se inicializan en cero o uno, dependiendo de su uso (presentaremos ejemplos que muestran cada uno de estos usos). Una variable que no se inicializa contiene *valores "basura"*; es decir, el último valor almacenado en la ubicación de memoria reservada para dicha variable.



**Error común de programación 3.5**

*Si no se inicializa un contador o un total, probablemente los resultados de su programa serán incorrectos. Éste es un ejemplo de un error de lógica.*



**Tip para prevenir errores 3.2**

*Inicialice los contadores y los totales.*

Observe que el cálculo del promedio dentro del programa produce un resultado igual a 81. En realidad, la suma de las calificaciones en este ejemplo es igual a 817 el cual, al dividirse entre 10 debe arrojar 81.7, es decir, un número con un punto decimal. En la siguiente sección veremos cómo manejar dichos tipos de números (llamados números de punto flotante).

## 3.9 Formulación de algoritmos mediante mejoramiento arriba-abajo, paso a paso: Ejemplo práctico 2 (repetición controlada por centinela)

Generalicemos el problema del promedio del grupo. Considere el siguiente problema:

*Desarrolle un programa de promedios de un grupo que procese un número arbitrario de calificaciones cada vez que se ejecute el programa.*

En el primer ejemplo del promedio del grupo, ya conocíamos previamente el número de calificaciones (10). En este ejemplo, no se indica cuántos datos se van a introducir. El programa debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo detener la introducción de los datos? ¿Cómo saber cuándo calcular e imprimir el promedio del grupo?

Una manera de resolver este problema es utilizar un valor especial llamado *valor centinela* (también conocido como *valor de señal*, *valor falso*, o *valor de bandera*) para indicar el "fin de la entrada de datos". El usuario introduce las calificaciones mientras sean valores legítimos. Entonces, el usuario introduce el valor centinela para indicar que ya se introdujo el último valor. A menudo, a la repetición controlada por centinela se le llama *repetición indefinida*, debido a que no se conoce el número de repeticiones antes de que comience la ejecución del ciclo.

De manera clara, se debe elegir un valor que no se confunda con un valor de entrada legítimo. Dado que por lo general las calificaciones de un examen son números enteros no negativos, −1 es un valor centinela aceptable para este problema. Por lo tanto, la ejecución del programa del promedio del grupo puede procesar un flujo de entradas como 95, 96, 75, 74 y 89 y −1. Entonces, el programa calcularía e imprimiría el promedio del grupo con las calificaciones 95, 96, 75, 74 y 89 (−1 es el valor centinela, de manera que no debe entrar en el cálculo del promedio).

**Error común de programación 3.6**

*Elegir un valor centinela que también sea un valor legítimo.*

Resolvimos el programa del promedio de la clase mediante una técnica llamada *mejoramiento arriba-abajo, paso a paso*, una técnica que es esencial para desarrollar buenos programas estructurados. Comencemos con una representación en pseudocódigo de la *cima*:

*Determinar el promedio del grupo en un examen*

La cima, es una instrucción simple que describe la función general del programa. Como tal, es una representación completa del programa. Desafortunadamente, la cima rara vez describe con suficiente detalle al problema para poder escribirlo en C. Ahora, comencemos el proceso de mejoramiento. Dividamos la cima en una serie de tareas más pequeñas, las cuales mostraremos en el orden en el que requieren ejecutarse. Esto da como resultado el *primer mejoramiento*:

*Inicializa las variables*
*Introduce suma, y cuenta las calificaciones del examen*
*Calcula e imprime el promedio del grupo*

Aquí sólo hemos utilizado la estructura secuencial; los pasos mostrados se ejecutan en orden, uno después del otro.



**Observación de ingeniería de software 3.3**

*Cada mejoramiento, así como la cima misma, es una especificación completa del algoritmo; solamente varía el nivel de detalle.*

Para proseguir con el siguiente nivel de mejoramiento, es decir, el *segundo mejoramiento*, nos concentramos en variables específicas. Necesitamos el total de los números, la cuenta de cuántos números se procesaron, una variable que reciba un valor para cada calificación tal como se introduce y una variable que almacene el promedio calculado. La instrucción en pseudocódigo

*Inicializa las variables*

Se puede mejorar de la siguiente manera:

*Inicializa total en cero*
*Inicializa contador en cero*

Observe que sólo necesitamos inicializar total y contador; las variables promedio y calificación (para el promedio calculado y para la entrada de usuario, respectivamente) no lo requieren debido a que sus valores se sobrescribirán mediante el proceso de lectura destructiva que explicamos en el capítulo 2. La instrucción en pseudocódigo

*Introduce, suma y cuenta las calificaciones del examen*

requiere de una estructura de repetición (es decir, un ciclo) que introduzca de manera exitosa cada calificación. Dado que no sabemos de antemano cuántas calificaciones van a procesarse, utilizaremos una repetición controlada por centinela. El usuario introducirá calificaciones legítimas, una a la vez. Después de introducir la última calificación legítima, el usuario introducirá el valor centinela. El programa evaluará este valor después de que se introduzca cada calificación y terminará el ciclo cuando se introduzca el valor centinela. Entonces, el mejoramiento de la instrucción en pseudocódigo anterior es:

```
Introduce la primera calificación
While el usuario no introduzca el centinela
    Suma esta calificación al total
    Suma uno al contador de calificaciones
    Introduce la siguiente calificación (posiblemente el centinela)
```

Observe que en pseudocódigo no necesitamos utilizar llaves alrededor de un conjunto de instrucciones que forman el cuerpo de una instrucción *while*. Simplemente colocamos una sangría en todas las instrucciones bajo *while* para indicar que pertenecen a *while*. De nuevo, el pseudocódigo es solamente una herramienta para desarrollar programas.

La instrucción en pseudocódigo

*Calcula e imprime el promedio del grupo*

se puede definir de la siguiente manera:

```
if el contador no es igual que cero
    Establece el promedio con el total dividido entre el contador
    Imprime el promedio
else
    Imprime "No se introdujeron calificaciones"
```

Observe que estamos siendo cuidadosos al considerar la posibilidad de una división entre cero, un *error fatal* que si no se detecta podría ocasionar que el programa fallara (a éste, a menudo se le llama "*estallamiento*" o "*estrellamiento*"). En la figura 3.7 mostramos el segundo mejoramiento.

**Error común de programación 3.7**

*Intentar una división entre cero ocasiona un error fatal.*



**Buena práctica de programación 3.5**

*Cuando realice divisiones con expresiones cuyo denominador pueda ser cero, haga una prueba explícita de este caso y manéjela de manera apropiada dentro de su programa (tal como la impresión de un mensaje de error), en lugar de permitir que ocurra un error fatal.*

En las figuras 3.5 y 3.6, dentro del pseudocódigo incluimos algunas líneas en blanco para mayor claridad. En realidad, las líneas en blanco separan al programa en sus distintas fases.

```
Inicializa total en cero
Inicializa contador en cero

Introduce la primera calificación
While el usuario no introduzca el centinela
    Suma esta calificación al total
    Suma uno al contador de calificaciones
    Introduce la siguiente calificación (posiblemente el centinela)

if el contador no es igual que cero
    Establece el promedio con el total dividido entre el contador
    Imprime el promedio
else
    Imprime "No se introdujeron calificaciones"
```

**Figura 3.7** Algoritmo en pseudocódigo que utiliza una repetición controlada por centinela para resolver el problema del promedio de un grupo.



### Observación de ingeniería de software 3.4

*Muchos programas pueden dividirse de manera lógica en tres fases: una fase de inicialización que especifica el valor inicial de las variables del programa; una fase de procesamiento que introduce los valores de los datos y ajusta las variables del programa de acuerdo con ello; y una fase de terminación que calcula e imprime los resultados finales.*

El algoritmo en pseudocódigo de la figura 3.7 resuelve el problema más general del promedio de un grupo. Este algoritmo se desarrolló después de sólo dos pasos de mejoramiento. Algunas veces se requieren más niveles.

### Observación de ingeniería de software 3.5

*El programador termina el proceso de mejoramiento arriba-abajo, paso a paso cuando el algoritmo en pseudocódigo se especifica con el detalle suficiente para que pueda convertir el pseudocódigo a C. Por lo general, la implementación del programa en C es directa.*

En la figura 3.8 mostramos el programa en C y una ejecución de ejemplo. Aunque sólo se introduzcan números enteros, es muy probable que el cálculo del promedio produzca un número con un punto decimal. El tipo **`int`** no puede representar dicho número. El programa introduce el tipo de dato **`float`** para manipular números con puntos decimales (llamados *números de punto flotante*), e introduce un operador especial llamado *operador de conversión de tipo* para manipular el cálculo del promedio. Explicaremos estas características con detalle, después de presentar el programa.

```
/* Figura 3.8: fig03_08.c
   Programa para obtener el promedio de calificaciones de un grupo mediante
   una repetición controlada por centinela */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   int contador;     /* número de calificaciones introducidas */
   int calificacion; /* valor de la calificación */
   int total;        /* suma de las calificaciones */

   float promedio;   /* número con punto decimal para el promedio */

   /* fase de inicialización */
   total    = 0; /* inicializa el total */
   contador = 0; /* inicializa el contador del ciclo */

   /* fase de procesamiento */
   /* obtiene la primera calificación del usuario */
   printf( “Introduzca la calificacion, -1 para terminar: “ );
   /* indicador para la entrada */
   scanf( “%d”, &calificacion ); /* lee la calificación del usuario */

   /* repite el ciclo mientras no se introduzca el valor centinela */
   while ( calificacion != -1 ) {
      total = total + calificacion; /* suma calificación a total */
      contador = contador + 1;      /* incrementa el contador */

      /* obtiene la siguiente calificación del usuario */
      printf( “Introduzca la calificacion, -1 para terminar: “ );
      /* indicador para la entrada */
      scanf(“%d”, &calificacion);       /* lee la siguiente calificación */
   } /* fin de while */
```

**Figura 3.8** Programa en C y ejecución de ejemplo del problema correspondiente al promedio del grupo mediante una repetición controlada por centinela. (Parte 1 de 2.)

```
32
33     /* fase de terminación */
34     /* si el usuario introdujo al menos una calificación */
35     if ( contador != 0 ) {
36
37        /* calcula el promedio de todas las calificaciones introducidas */
38        promedio = ( float ) total / contador; /* evita que se trunque*/
39
40        /* despliega el promedio con dos dígitos de precisión */
41        printf( " El promedio del grupo es: %.2f\n", promedio );
42     } /* fin de if*/
43     else { /* si no se introdujo calificación alguna, despliega el mensaje */
44        printf( "No se introdujeron calificaciones\n" );
45     } /* fin de else */
46
47     return 0; /* indica que el programa terminó con éxito */
48
49  } /* fin de la función main */
```

```
Introduzca la calificacion, -1 para terminar: 75
Introduzca la calificacion, -1 para terminar: 94
Introduzca la calificacion, -1 para terminar: 97
Introduzca la calificacion, -1 para terminar: 88
Introduzca la calificacion, -1 para terminar: 70
Introduzca la calificacion, -1 para terminar: 64
Introduzca la calificacion, -1 para terminar: 83
Introduzca la calificacion, -1 para terminar: 89
Introduzca la calificacion, -1 para terminar: -1
El promedio del grupo es: 82.50
```

```
Introduzca la calificacion, -1 para terminar: -1
No se introdujeron calificaciones
```

**Figura 3.8** Programa en C y ejecución de ejemplo del problema correspondiente al promedio del grupo mediante una repetición controlada por centinela. (Parte 2 de 2.)

En la figura 3.8, observe la instrucción compuesta dentro del ciclo **`while`** (línea 24). De nuevo, las llaves son necesarias para las cuatro instrucciones que se van a ejecutar dentro del ciclo. Sin las llaves, las últimas tres instrucciones del cuerpo del ciclo estarían fuera de éste, lo que provocaría que la computadora interpretara este código de manera incorrecta, como lo mostramos a continuación:

```
while ( calificacion != -1 )
   total = total + calificacion;       /* suma calificación a total */
contador = contador + 1;               /* incrementa el contador */
printf( "Introduzca la calificacion,   /* indicador para la entrada */
        -1 para terminar: " );
scanf("%d", &calificacion);            /* lee la siguiente calificación */
```

Si el usuario no introduce **`-1`** como primera calificación, esto provocaría un ciclo infinito.



**Buena práctica de programación 3.6**

*En un ciclo controlado por centinela, la indicación de entrada de datos debe recordar de manera explícita cuál es el valor del centinela.*

Los promedios no siempre arrojan números enteros. A menudo, un promedio es un valor como 7.2 o −93.5, los cuales contienen una parte fraccional. A estos valores se les conoce como números de punto flotante

y se representan mediante el tipo de dato **float**. La variable promedio se define como de tipo **float** (línea 12) para capturar el resultado fraccional de nuestro cálculo. Sin embargo, el resultado del cálculo **total / contador** es un entero, debido a que total y contador son variables enteras. Al dividir dos enteros obtenemos como resultado una *división entera*, en la cual se pierde cualquier parte fraccional (es decir, *se trunca*). Dado que primero se realiza el cálculo, la parte fraccional se pierde antes de que el resultado se asigne a promedio. Para producir un cálculo con formato de punto flotante mediante valores enteros, debemos crear valores temporales que sean números de punto flotante. C proporciona el *operador unario de conversión de tipo* para llevar a cabo esta tarea. La línea 38

```
promedio = ( float ) total / contador;
```

incluye el operador de conversión de tipo **(float)**, el cual crea una copia temporal de su operando, como número de punto flotante, llamada **total**. El valor almacenado en **total** permanece como un entero. Al hecho de utilizar el operador de conversión de esta manera, se le llama *conversión explícita*. El cálculo consiste ahora en un valor de punto flotante (la versión **float** de **total**) dividido entre el valor entero almacenado en **contador**. El compilador de C sabe cómo evaluar las expresiones sólo si los tipos de datos de los operandos son idénticos. Para garantizar que los operandos sean del mismo tipo, el compilador realiza una operación llamada *promoción* (o *conversión implícita*) de los operadores seleccionados. Por ejemplo, en una expresión que contiene datos de tipo **int** y **float**, se hacen copias de los operandos **int** y se *promueven* a **float**. En nuestro ejemplo, después de hacer una copia de contador y promoverlo a **float**, se realiza el cálculo y el resultado de la división de números de punto flotante se asigna a promedio. En el capítulo 5, presentaremos una explicación de todos los tipos de datos estándar y su orden de promoción.

Los operadores de conversión de tipo están disponibles para la mayoría de los tipos de datos. El operador de conversión de tipo es un *operador unario*, es decir, un operador que toma sólo un operando. En el capítulo 2, estudiamos los operadores aritméticos binarios. C también permite las versiones unarias de los operadores de suma (+) y de resta (−), de tal modo que los programadores pueden escribir expresiones como **-7** o **+5**. Los operadores de conversión de tipo se asocian de derecha a izquierda y tienen la misma precedencia que otros operadores unarios tales como el + unario o el – unario. Esta precedencia es un nivel más alto que la de los *operadores de multiplicación* como *****, **/** y **%**.

En la figura 3.8 utilizamos el especificador de conversión de **printf**, **%.2f** (línea 41), para imprimir el valor del **promedio**. La **f** especifica que se imprimirá un valor de punto flotante. El **.2** es la *precisión* con la cual se desplegará el valor; es decir, el valor se desplegará con 2 dígitos a la derecha del punto decimal. Si se utiliza el especificador de conversión **%f** (sin especificar la precisión), se utiliza una *precisión predeterminada* de 6, exactamente como si se hubiera utilizado **%.6f**. Cuando los valores de punto flotante se imprimen con precisión, el valor impreso se *redondea* al número indicado de posiciones decimales. El valor en memoria se mantiene inalterado. Cuando se ejecutan las siguientes instrucciones,

```
printf( "%.2f\n", 3.446); /* imprime 3.45 */
printf( "%.1f\n", 3.446); /* imprime 3.4 */
```

se imprimen los valores 3.45 y 3.4.

**Error común de programación 3.8**

*Utilizar precisión en una especificación de conversión dentro de la cadena de control de formato de la instrucción **scanf** es un error. Las precisiones se utilizan solamente en las especificaciones de conversión de **printf**.*

**Error común de programación 3.9**



*Utilizar números de punto flotante de manera que se asuma una representación precisa, puede provocar resultados incorrectos. En la mayoría de las computadoras, los números de punto flotante se representan únicamente de manera aproximada.*



**Tip para prevenir errores 3.3**

*No compare la igualdad de valores de punto flotante.*

A pesar de que los números de punto flotante no siempre son "100% precisos", tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de la temperatura "normal" de 36.5, no necesitamos precisar un largo número de dígitos. Cuando vemos la temperatura en un termómetro y leemos que es igual a 36.5, en realidad

podría ser 36.5985999473210643. El punto aquí es que llamar al número simplemente 36.5 es correcto para la mayoría de las aplicaciones. Posteriormente ahondaremos más en este tema.

Otra manera de generar números de punto flotante es a través de la división. Cuando dividimos 10 entre 3, el resultado es 3.3333333…, en donde la secuencia de números 3 se repite de manera indefinida. La computadora reserva un espacio fijo para almacenar dicho valor, de manera que se aprecia claramente que el número de punto flotante solamente puede almacenar una aproximación.

## 3.10 Formulación de algoritmos mediante mejoramiento arriba-abajo, paso a paso: Ejemplo práctico 3 (estructuras de control anidadas)

Trabajemos en otro programa funcional. De nuevo, formularemos el algoritmo en pseudocódigo mediante el mejoramiento paso a paso, de arriba abajo, y escribiremos el programa en C correspondiente. Hemos visto que las instrucciones de control se pueden apilar una arriba de la otra (en secuencia), tal como un niño apila bloques de construcción. En este ejemplo práctico veremos la otra manera en que se pueden conectar las instrucciones de control, a saber, a través del *anidamiento* de una instrucción de control dentro de otra.

Considere el siguiente enunciado del problema:

*Un colegio ofrece un curso que prepara a los estudiantes para el examen estatal con el que se obtiene la certificación como corredor de bienes raíces. El año pasado, muchos de los estudiantes que completaron el curso tomaron el examen de certificación. De manera natural, el colegio desea saber qué tan bien se desenvuelven los estudiantes en el examen. A usted se le pide que escriba un programa para sumar los resultados. Para comenzar, se le proporciona una lista de estos diez estudiantes. Junto a cada nombre se escribe un 1 si el estudiante pasó el examen y un 2 si el estudiante lo reprobó.*

*Su programa debe analizar los resultados del examen de la siguiente manera:*

1. *Introduzca los resultados del examen (es decir, 1 o 2). En la pantalla, despliegue el mensaje "Introduzca resultado", cada vez que el programa solicite otro resultado de examen.*
2. *Cuente el número de resultados de cada tipo.*
3. *Despliegue un resumen de los resultados del examen, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.*
4. *Si aprobaron el examen más de ocho estudiantes, imprima el mensaje "Se logró el objetivo".*

Después de leer cuidadosamente el enunciado del problema, haremos las siguientes observaciones:

1. El programa debe procesar 10 resultados de examen. Utilizaremos un ciclo controlado por contador.
2. Cada resultado del examen es un número, un 1 o un 2. Cada vez que el programa lee un resultado de examen, el programa debe determinar si el número es un 1 o un 2. En nuestro programa, evaluamos un 1. Si el número no es un 1, asumimos que es un 2. (Un ejercicio al final del capítulo considera las consecuencias de asumir lo anterior).
3. Se utilizan dos contadores, uno para contar el número de estudiantes que aprobaron el examen y otro para contar el número de estudiantes que lo reprobaron.
4. Una vez que el programa ha procesado todos los resultados, éste debe decidir si aprobaron más de 8 estudiantes.

Procedamos con el mejoramiento arriba-abajo, paso a paso. Comenzamos con la representación en pseudocódigo de la cima:

*Analiza los resultados del examen y decide si se logra el objetivo*

Una vez más, es importante enfatizar que la cima es una representación completa del programa, pero muy probablemente se requerirán muchos mejoramientos antes de que el pseudocódigo evolucione de manera natural a un programa en C. Nuestro primer mejoramiento es:

*Inicializa las variables*
*Introduce las diez calificaciones del examen y cuenta el número de aprobados y reprobados*
*Imprime un resumen de los resultados del examen y decide si se cumplió el objetivo del curso*

Aquí también, a pesar de que tenemos una representación completa del programa, requerimos un mayor mejoramiento. Ahora nos concentramos en las variables específicas. Los contadores son necesarios para registrar los aprobados y los reprobados; utilizaremos un contador para controlar el proceso del ciclo; y necesitamos una variable para almacenar la entrada del usuario. La instrucción en pseudocódigo

*Inicializa las variables*

puede mejorarse de la siguiente manera

*Inicializa aprobados en cero*
*Inicializa reprobados en cero*
*Inicializa contador estudiante en uno*

Observe que sólo se inicializan los contadores. La instrucción en pseudocódigo

*Introduce las diez calificaciones del examen y cuenta el número de aprobados y reprobados*

requiere que un ciclo introduzca de manera exitosa los resultados de cada examen. Aquí sabemos por anticipado que existen exactamente 10 resultados del examen, de manera que es apropiado un ciclo controlado por contador. Dentro del ciclo (es decir, *anidada* dentro de él), una instrucción de selección doble determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, el mejoramiento de la instrucción en pseudocódigo anterior es

```
While contador estudiante sea menor o igual que diez
    Introduce el siguiente resultado de examen

    If el estudiante aprobó
        Suma uno a contador aprobados
    else
        Suma uno a contador reprobados

        Suma uno al contador estudiante
```

Observe que utilizamos líneas en blanco para resaltar la instrucción *if…else* y mejorar la claridad del programa. La instrucción en pseudocódigo

*Imprime un resumen de los resultados del examen y decide si se cumplió el objetivo del curso*

podría mejorarse de la siguiente manera:

```
Imprime el número de aprobados
Imprime el número de reprobados
    Si aprobaron más de ocho estudiantes
    imprime "Objetivo cumplido"
```

El segundo mejoramiento completo aparece en la figura 3.9. Observe que las líneas en blanco también se utilizan para resaltar la instrucción *while* y mejorar la claridad de los programas.

Ahora, el pseudocódigo está suficientemente mejorado para convertirlo al programa en C. La figura 3.10 muestra el programa en C y las dos ejecuciones de ejemplo. Observe que aprovechamos la característica de C que nos permite que la inicialización se incorpore en las definiciones. Dicha inicialización ocurre en tiempo de compilación.

```
Inicializa aprobados en cero
Inicializa reprobados en cero
Inicializa contador estudiante en uno

While contador estudiante sea menor o igual que diez
        Introduce el siguiente resultado de examen
```

**Figura 3.9** Pseudocódigo para el problema de los resultados del examen. (Parte 1 de 2.)

*If el estudiante aprobó*
*Suma uno a contador aprobados*
*else*
*Suma uno a contador reprobados*

*Suma uno a contador estudiante*

*Imprime el número de aprobados*
*Imprime el número de reprobados*
*Si aprobaron más de ocho estudiantes*
*imprime “Objetivo cumplido”*

**Figura 3.9** Pseudocódigo para el problema de los resultados del examen. (Parte 2 de 2.)

```
/* Figura 3.10: fig03_10.c
   Análisis de los resultados de un examen */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   /* inicializa las variables en las declaraciones */
   int aprobados = 0;   /* número de aprobados */
   int reprobados = 0;  /* número de reprobados*/
   int estudiante = 1;  /* contador de estudiantes */
   int resultado;       /* resultado de un examen */

   /* procesa 10 estudiantes mediante un ciclo controlado por contador */
   while ( estudiante <= 10 ) {

      /* indica al usuario que introduzca un valor */
      printf( “Introduzca el resultado ( 1=aprobado,2=reprobado ): “ );
      scanf( “%d”, &resultado );

      /* si el resultado es igual que 1, incrementa aprobados */
      if ( resultado == 1 ) {
         aprobados = aprobados + 1;
      } /* fin de if */
      else { /* de lo contrario, incrementa reprobados */
         reprobados = reprobados + 1;
      } /* fin de else */

      estudiante = estudiante + 1; /* incrementa el contador de estudiante */
   } /* fin de while */

   /* fase de terminación; despliega el número de aprobados y reprobados */
   printf( “Aprobados %d\n”, aprobados );
   printf( “Reprobados %d\n”, reprobados );

   /* si aprobaron más de ocho estudiantes, imprime “objetivo alcanzado” */
   if ( aprobados > 8 ) {
      printf( “Objetivo alcanzado\n” );
```

**Figura 3.10** Programa en C y ejecuciones de muestra para el problema de los resultados del examen. (Parte 1 de 2.)

```
    } /* fin de if */

    return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

```
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 2
Introduzca el resultado ( 1=aprobado,2=reprobado ): 2
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 2
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 2
Aprobados 6
Reprobados 4
```

```
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 2
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Introduzca el resultado ( 1=aprobado,2=reprobado ): 1
Aprobados 9
Reprobados 1
Objetivo alcanzado
```

**Figura 3.10** Programa en C y ejecuciones de muestra para el problema de los resultados del examen. (Parte 2 de 2.)

**Tip de rendimiento 3.1**

*Inicializar variables al momento de declararlas puede ayudar a reducir el tiempo de ejecución de un programa.*



**Tip de rendimiento 3.2**

*Muchos de los tips de rendimiento que escribimos en este libro provocan mejoras mínimas, de manera que el lector podría verse tentado a ignorarlas. Observe que el efecto acumulado de todas estas mejoras de rendimiento puede hacer que el rendimiento del programa mejore de manera significativa. Además, se puede apreciar una mejora importante cuando se refina un poco un ciclo que se repite un gran número de veces.*



**Observación de ingeniería de software 3.6**

*La experiencia ha demostrado que la parte más difícil para solucionar un problema en una computadora es el desarrollo del algoritmo de dicha solución. Por lo general, una vez que se especifica un algoritmo correcto, el proceso para producir un programa en C es directo.*



**Observación de ingeniería de software 3.7**

*Muchos programadores escriben programas sin utilizar herramientas de diseño de programas tales como pseudocódigo. Ellos sienten que su meta final es la de resolver el problema en la computadora y que escribir pseudocódigo solamente retrasa la producción del resultado final.*

| Operador de asignación | Expresión de ejemplo | Explicación | Asigna |
|---|---|---|---|
| *Suponga que*: **`int c = 3, d = 5, e = 4, f = 6, g = 12;`** | | | |
| `+=` | `c += 7` | `c = c + 7` | `10` a `c` |
| `-=` | `d -= 4` | `d = d - 4` | `1` a `d` |
| `*=` | `e *= 5` | `e = e * 5` | `20` a `e` |
| `/=` | `f /= 3` | `f = f / 3` | `2` a `f` |
| `%=` | `g %= 9` | `g = g % 9` | `3` a `g` |

**Figura 3.11** Operadores aritméticos de asignación.

## 3.11 Operadores de asignación

C proporciona varios operadores de asignación para abreviar las expresiones de asignación. Por ejemplo, la instrucción

```
c = c + 3;
```

se puede abreviar mediante el *operador de asignación de suma* **`+=`** como

```
c += 3;
```

El operador **`+=`** suma el valor de la expresión que se encuentra a la derecha del operador, al valor de la variable que se encuentra a la izquierda del operador y almacena el resultado en la variable que está a la izquierda del operador. Cualquier instrucción de la forma

*variable = variable operador expresión;*

en donde el *operador* es uno de los operadores binarios **`+`**, **`-`**, **`*`**, **`/`** o **`%`** (u otros que explicaremos en el capítulo 10), se pueden escribir en la forma

*variable operador= expresión;*

Por lo tanto, la asignación **`c += 3`** suma **`3`** a **`c`**. La figura 3.11 muestra los operadores aritméticos de asignación, expresiones de ejemplo que utilizan estos operadores, y explicaciones.

## 3.12 Operadores de incremento y decremento

C también proporciona el operador unario de *incremento,* **`++`**, y el operador unario de *decremento,* **`--`**, los cuales se resumen en la figura 3.12. Si la variable **`c`** se incrementa en 1, podemos utilizar el operador de incremento **`++`**, en lugar de las expresiones **`c = c + 1`** o **`c+= 1`**. Si los operadores de incremento o decremento se colocan antes de una variable, se les llama *operadores de preincremento o predecremento* respectivamente. Si los operadores de incremento y decremento se colocan después de la variable, se les llama *operadores de posincrement y posdecremento* respectivamente. Preincrementar (predecrementar) una variable provoca que la variable se incremente (decremente) en 1, y después el nuevo valor de la variable se utiliza en la expresión en la cual aparece. Posincrementar (posdecrementar) la variable provoca que el valor actual de la variable se utilice en la expresión en la que aparece, y después el valor de la variable se incrementa (decrementa) en 1.

| Operador | Expresión de ejemplo | Explicación |
|---|---|---|
| `++` | `++a` | Incrementa **`a`** en 1 y después utiliza el nuevo valor de **`a`** en la expresión en la que reside. |
| `++` | `a++` | Utiliza el valor actual de **`a`** en la expresión en la que reside, y después la incrementa en 1. |

**Figura 3.12** Operadores de incremento y decremento. (Parte 1 de 2.)

| Operador | Expresión de ejemplo | Explicación |
|---|---|---|
| `--` | `--b` | Decrementa **b** en 1 y después utiliza el nuevo valor de **b** en la expresión en la cual reside. |
| `--` | `b--` | Utiliza el valor actual de **b** en la expresión en la cual reside **b**, y después decrementa **b** en 1. |

**Figura 3.12** Operadores de incremento y decremento. (Parte 2 de 2.)

```
/* Figura 3.13: fig03_13.c
    Preincremeno y posincremento */
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   int c;              /* define la variable */

   /* demuestra el posincremento */
   c = 5;                 /* le asigna 5 a c */
   printf( "%d\n", c );   /* imprime 5 */
   printf( "%d\n", c++ ); /* imprime 5 y hace el posincremento */
   printf( "%d\n\n", c ); /* imprime 6 */

   /* demuestra el preincremento */
   c = 5;                 /* le asigna 5 a c */
   printf( "%d\n", c );   /* imprime 5 */
   printf( "%d\n", ++c ); /* preincrementa y después imprime 6 */
   printf( "%d\n", c );   /* imprime 6 */

   return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

```
5
5
6

5
6
6
```

**Figura 3.13** Preincremento en comparación con posincremento.

La figura 3.13 muestra la diferencia entre las versiones de preincremento y posincremento del operador **`++`**. Posincrementar la variable **`c`** provoca que ésta se incremente después de utilizarla en la instrucción **`printf`**. Preincrementar la variable **`c`** provoca que ésta se incremente antes de utilizarla en la instrucción **`printf`**.

El programa despliega el valor de **`c`** antes y después de que se utilice el operador **`++`**. El operador de decremento (**`--`**) funciona de manera similar.



**Buena práctica de programación 3.7**

*Los operadores unarios deben colocarse inmediatamente después de sus operandos, sin espacios intermedios.*

Las tres instrucciones de asignación de la figura 3.10

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
estudiante = estudiante + 1;
```

pueden escribirse de manera más concisa mediante operadores de asignación como

```
aprobados += 1;
reprobados += 1;
estudiante += 1;
```

con los operadores de preincremento como

```
++aprobados;
++reprobados;
++estudiante;
```

o con operadores de posincremento como

```
aprobados++;
reprobados++;
estudiante++;
```

Aquí, es importante observar que cuando se incrementa o decrementa por sí misma una variable dentro de una instrucción, las formas de preincremento y posdecremento tienen el mismo efecto. Es sólo cuando la variable aparece en el contexto de una expresión más grande que el preincremento y el posdecremento tienen efectos diferentes (similar para el predecremento y el posdecremento). Sólo un nombre de variable simple puede utilizarse como operando de un operador de incremento o decremento.



**Error común de programación 3.10**

*Intentar utilizar el operador de incremento o decremento en una expresión que no sea un nombre de variable simple es un error de sintaxis; por ejemplo,* **`++(x + 1)`**.



**Tip para prevenir errores 3.4**

*Por lo general, C no especifica el orden en que se evaluarán los operandos del operador (aunque en el capítulo 4 veremos excepciones para unos cuantos operadores). Por lo tanto, el programador debe evitar el uso de instrucciones con operadores de incremento y decremento en las que una variable que se incrementa o decrementa aparece más de una vez.*

La figura 3.14 muestra la precedencia y asociatividad de los operadores que hemos presentado hasta este punto. Los operadores aparecen en orden decreciente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. Observe que el operador condicional (**?:**), los operadores unarios de incremento (**++**), decremento (**--**), suma (**+**), menos (**-**), de conversión de flujo, y los operadores de asignación **=**, **+=**, **-=**, ***=**, **/=** y **%=** se asocian de derecha a izquierda. La tercera columna especifica los distintos grupos de operadores. Todos los demás operadores de la figura 3.14 se asocian de izquierda a derecha.

| Operadores | Asociatividad | Tipo |
|---|---|---|
| `++ -- + -` (*tipo*) | derecha a izquierda | unario |
| `* / %` | izquierda a derecha | multiplicativo |
| `+ -` | izquierda a derecha | aditivo |
| `< <= > >=` | izquierda a derecha | de relación |
| `== !=` | izquierda a derecha | de igualdad |
| `?:` | derecha a izquierda | condicional |
| `= += -= *= /= %=` | derecha a izquierda | de asignación |

**Figura 3.14** Precedencia de los operadores tratados hasta este punto del texto.

## *RESUMEN*

- La solución de cualquier problema de computación involucra una serie de acciones en un orden específico. Al procedimiento para resolver un problema en términos de las acciones que se van a ejecutar y el orden en el que dichas acciones se deben ejecutar se le llama algoritmo.
- A la especificación del orden en el cual se van a ejecutar las instrucciones dentro de un programa se le llama control del programa.
- El pseudocódigo es un lenguaje artificial e informal que ayuda a los programadores a desarrollar algoritmos. Es similar al idioma inglés. En realidad, los programas en pseudocódigo no se ejecutan en las computadoras; solamente ayuda al programador a "plantear" un programa antes de intentar escribirlo en un lenguaje de programación tal como C.
- El pseudocódigo solamente consiste en caracteres, de manera que los programadores pueden teclear programas en pseudocódigo dentro de la computadora, editarlos, y guardarlos.
- El pseudocódigo consiste solamente en instrucciones ejecutables. Las declaraciones son mensajes para el compilador, para indicarle las características de las variables y reservar espacio para éstas.
- Una instrucción de selección se utiliza para elegir entre distintos cursos de acción.
- La instrucción de selección **`if`** ejecuta la acción indicada solamente si la condición es verdadera.
- La instrucción de selección **`if…else`** especifica la ejecución de acciones por separado: cuando la condición es verdadera y cuando la condición es falsa.
- Una instrucción de selección **`if…else`** anidada puede evaluar muchos casos diferentes. Si más de una condición es verdadera, solamente se ejecutarán las instrucciones que se encuentran después de la primera condición verdadera.
- Siempre que se vaya a ejecutar más de una instrucción, en donde por lo general se coloca sólo una, dichas instrucciones deben encerrarse entre llaves para formar una instrucción compuesta. Una instrucción compuesta puede colocarse en cualquier parte donde se pueda colocar una instrucción simple.
- Una instrucción vacía, que indica que no se realizará acción alguna, se establece mediante un punto y coma (**`;`**) en donde normalmente iría una instrucción.
- Una instrucción de repetición especifica que una acción se repetirá mientras cierta condición sea verdadera.
- La instrucción (o instrucción compuesta o bloque) contenida en la instrucción de repetición **`while`** constituye el cuerpo del ciclo.
- Por lo general, alguna instrucción especificada dentro del cuerpo de una instrucción **`while`**, en algún momento modificará la condición para que sea falsa. De lo contrario, el ciclo nunca terminará; un error conocido como ciclo infinito.
- El ciclo controlado por contador utiliza una variable como un contador para determinar cuándo debe terminar el ciclo.
- Un total es una variable que acumula la suma de una serie de números. Por lo general, los totales se inicializan en cero antes de la ejecución del programa.
- Un diagrama de flujo es una representación gráfica de un algoritmo. Los diagramas de flujo se dibujan utilizando ciertos símbolos especiales como óvalos, rectángulos, rombos, y pequeños círculos conectados mediante flechas llamadas líneas de flujo. Los símbolos indican las acciones a realizar. Las líneas de flujo indican el orden en el que se realizan las acciones.
- El símbolo óvalo, también llamado símbolo de terminación, indica el inicio y el final de cada algoritmo.
- El símbolo rectángulo, también llamado símbolo de acción, indica cualquier tipo de cálculo u operación de entrada/salida. Por lo general, los símbolos rectángulos corresponden a las acciones que realizan las instrucciones de asignación, o a las operaciones de entrada/salida que normalmente llevan a cabo funciones de la biblioteca estándar como **`printf`** y **`scanf`**.
- El símbolo rombo, también llamado símbolo de decisión, indica que se tomará una decisión. El símbolo de decisión contiene una expresión que puede ser falsa o verdadera. Dos líneas de flujo emergen de él. Una línea de flujo indica la dirección que se debe tomar cuando la condición es verdadera; la otra indica la dirección que se debe tomar cuando la condición es falsa.
- A un valor que contiene una parte fraccional se le conoce como número de punto flotante y se representa mediante el tipo de dato **`float`**.
- Cuando se dividen dos enteros, se pierde la parte fraccionaria del cálculo (es decir, se trunca).
- C proporciona el operador unario de conversión de tipo (**`float`**) para crear una copia de punto flotante de su operando. Al uso de un operador de conversión de tipo se le llama conversión explícita. Los operadores de conversión de flujo están disponibles para la mayoría de los tipos de datos.

- El compilador de C sabe cómo evaluar expresiones, sólo cuando los tipos de los operandos son idénticos. Para asegurarse de que los operandos sean del mismo tipo, el compilador realiza una operación llamada promoción (también conocida como conversión implícita) sobre los operandos seleccionados. En particular, los operandos **`int`** se promueven a **`float`**. C proporciona un conjunto de reglas para la promoción de operandos de tipos diferentes.
- Los valores de punto flotante aparecen con un número específico de dígitos después del punto decimal cuando se especifica una precisión con el especificador de precisión **`%f`** dentro de una instrucción **`printf`**. El valor **`3.456`** aparece como **`3.46`** cuando se le aplica el especificador de conversión **`%.2f`**. Si utilizamos el especificador de conversión **`%f`** (sin especificar la precisión), se utiliza la precisión predeterminada 6.
- C proporciona varios operadores de asignación que ayudan a abreviar ciertos tipos comunes de expresiones de asignación. Estos operadores son: **`+=`**, **`-=`**, **`*=`**, **`/=`**, y **`%=`**. En general, cualquier instrucción de la forma

  *Variable = variable operador expresión;*

  donde **`operador`** es uno de los operadores **`+`**, **`-`**, **`*`**, **`/`** o **`%`**, se puede escribir de la forma

  *Variable operador= expresión;*
- C proporciona el operador de incremento, **`++`** y el operador de decremento **`--`**, para incrementar o decrementar una variable en 1. Estos operadores se pueden colocar como prefijo o sufijo de una variable. Si el operador se coloca como prefijo de la variable, ésta incrementa primero en 1, y luego se utiliza en su expresión. Si el operador se coloca como sufijo de la variable, ésta se utiliza dentro de su expresión, y luego se incrementa o decrementa en 1.

## TERMINOLOGÍA

acción
algoritmo
bloque
caracteres blancos
ciclo infinito
cima
condición de terminación
contador
control de programa
conversión explícita
conversión implícita
cuerpo de un ciclo
decisión
diagrama de flujo
división entera
división entre cero
ejecución secuencial
eliminación de **`goto`**
error de sintaxis
error fatal
error lógico
errores no fatales
"estallamiento"
"estrellamiento"
estructura de control
estructura de control de entrada simple/salida simple
estructura de secuencia
estructuras de control anidadas
estructuras de control apiladas
estructuras de repetición
estructuras de selección
expresión condicional
fase de inicialización
fase de procesamiento
fase de terminación
"fin de la entrada de datos"
**`float`**
inicialización
instrucción compuesta
instrucción de repetición **`while`**
instrucción de selección doble
instrucción de selección **`if`**
instrucción de selección **`if…else`**
instrucción de selección múltiple
instrucción de selección simple
instrucción **`goto`**
instrucción vacía (**`;`**)
instrucciones **`if…else`** anidadas
línea de flujo
mejoramiento arriba abajo, paso a paso
mejoramiento paso a paso
número de punto flotante
operador condicional (**`?:`**)
operador de conversión de tipo
operador de decremento (**`--`**)
operador de incremento (**`++`**)
operador de posdecremento
operador de posincremento
operador de predecremento
operador de preincremento
operador ternario
operadores aritméticos de asignación: **`+=`**, **`-=`**, **`*=`**, **`/=`**, y **`%=`**
operadores de multiplicación
orden de las acciones
pasos
precisión
precisión predeterminada
primer mejoramiento
programación estructurada
promoción
pseudocódigo
redondeo
repetición
repetición controlada por contador
repetición definida
repetición indefinida
segundo mejoramiento
selección
símbolo de acción
símbolo de decisión
símbolo de diagrama de flujo
símbolo de fin
símbolo de óvalo
símbolo de terminación
símbolo rectángulo
símbolo rombo
símbolos conectores
símbolos de flecha
total
transferencia de control
truncar
valor "basura"
valor centinela
valor de bandera
valor de señal
valor falso

## ERRORES COMUNES DE PROGRAMACIÓN

**3.1** Olvidar una o las dos llaves que delimitan una instrucción compuesta.
**3.2** Colocar un punto y coma después de la condición de una instrucción **`if`** provoca un error de lógica dentro de las instrucciones **`if`** de selección simple, y un error de sintaxis en las instrucciones **`if`** de selección doble.
**3.3** No proporcionar una acción dentro del cuerpo de una instrucción **`while`** que permita que ésta se haga falsa, ocasionará que dicha estructura de repetición no termine nunca; a esto se le conoce como un "ciclo infinito".
**3.4** Escribir la palabra reservada **`while`** con una letra mayúscula, como en **`While`** (recuerde que C es un lenguaje sensible a mayúsculas y minúsculas). Todas las palabras reservadas de C tales como **`while`**, **`if`** y **`else`** contienen sólo letras minúsculas.
**3.5** Si no se inicializa un contador o un total, probablemente los resultados de su programa serán incorrectos. Éste es un ejemplo de un error de lógica.
**3.6** Elegir un valor centinela que también sea un valor legítimo.
**3.7** Intentar una división entre cero ocasiona un error fatal.
**3.8** Utilizar precisión en una especificación de conversión dentro de la cadena de control de formato de la instrucción **`scanf`** es un error. Las precisiones se utilizan solamente en las especificaciones de conversión de **`printf`**.
**3.9** Utilizar números de punto flotante de una manera que se asuma una representación precisa, puede provocar resultados incorrectos. En la mayoría de las computadoras, los números de punto flotante se representan únicamente de manera aproximada.
**3.10** Intentar utilizar el operador de incremento o decremento en una expresión que no sea un nombre de variable simple es un error de sintaxis; por ejemplo, **`++(x + 1)`**.

## TIPS PARA PREVENIR ERRORES

**3.1** Escribir las llaves inicial y final de instrucciones compuestas, antes de escribir las instrucciones individuales que van dentro de ellas, ayuda a evitar la omisión de una o ambas llaves, a prevenir errores de sintaxis y a prevenir errores de lógica (en donde se requieren ambas llaves).
**3.2** Inicialice los contadores y los totales.
**3.3** No compare la igualdad de valores de punto flotante.
**3.4** Por lo general, C no especifica el orden en que se evaluarán los operandos del operador (aunque en el capítulo 4 veremos excepciones para unos cuantos operadores). Por lo tanto, el programador debe evitar el uso de instrucciones con operadores de incremento y decremento en las que una variable que se incrementa o decrementa aparece más de una vez.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**3.1** La aplicación consistente de convenciones para el sangrado mejora de manera importante la claridad del programa. Le sugerimos un tabulador de tamaño fijo de 1/4 de pulgada o tres espacios en blanco por sangrado. En este libro, utilizamos tres espacios en blanco por sangrado.
**3.2** A menudo, el pseudocódigo se utiliza para "plantear" un programa durante el proceso de diseño. Posteriormente el programa en pseudocódigo se convierte a C.
**3.3** Coloque sangrías en las dos instrucciones que componen el cuerpo de una instrucción **`if…else`**.
**3.4** Si existen muchos niveles de sangrado, cada nivel debe estar sangrado con el mismo número de espacios.
**3.5** Cuando realice divisiones con expresiones cuyo denominador pueda ser cero, haga una prueba explícita de este caso y manéjela de manera apropiada dentro de su programa (tal como la impresión de un mensaje de error), en lugar de permitir que ocurra un error fatal.
**3.6** En un ciclo controlado por centinela, la indicación de entrada de datos debe recordar de manera explícita cuál es el valor del centinela.
**3.7** Los operadores unarios deben colocarse inmediatamente después de sus operandos, sin espacios intermedios.

## TIPS DE RENDIMIENTO

**3.1** Inicializar variables al momento de declararlas puede ayudar a reducir el tiempo de ejecución de un programa.
**3.2** Muchos de los tips de rendimiento que escribimos en este libro provocan mejoras mínimas, de manera que el lector podría verse tentado a ignorarlas. Observe que el efecto acumulado de todas estas mejoras de rendimiento pueden hacer que el rendimiento del programa mejore de manera significativa. Además, se puede apreciar una mejora importante cuando se refina un poco un ciclo que se repite un gran número de veces.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**3.1** Una instrucción compuesta puede colocarse en cualquier parte de un programa en donde pueda colocarse una instrucción sencilla.

**3.2** Tal como una instrucción compuesta puede colocarse en cualquier parte en donde coloque una instrucción sencilla, también es posible no tener instrucción alguna, es decir, tener una instrucción vacía. La instrucción vacía se representa colocando un punto y coma (**;**) en donde por lo general va la instrucción.

**3.3** Cada mejoramiento, así como la cima misma, es una especificación completa del algoritmo; solamente varía el nivel de detalle.

**3.4** Muchos programas pueden dividirse de manera lógica en tres fases: una fase de inicialización que especifica el valor inicial de las variables del programa; una fase de procesamiento que introduce los valores de los datos y ajusta las variables del programa de acuerdo con ello; y una fase de terminación que calcula e imprime los resultados finales.

**3.5** El programador termina el proceso de mejoramiento arriba-abajo, paso a paso cuando el algoritmo en pseudocódigo se especifica con el detalle suficiente para que el programador pueda convertir el pseudocódigo a C. Normalmente, la implementación del programa en C es directa.

**3.6** La experiencia ha demostrado que la parte más difícil para solucionar un problema en una computadora es el desarrollo del algoritmo para dicha solución. Por lo general, una vez que se especifica un algoritmo correcto, el proceso para producir un programa en C es directo.

**3.7** Muchos programadores escriben programas sin utilizar herramientas de diseño de programas tales como pseudocódigo. Ellos sienten que su meta final es la de resolver el problema en la computadora y que escribir pseudocódigo solamente retrasa la producción del resultado final.

## EJERCICIOS DE AUTOEVALUACIÓN

**3.1** Complete los espacios en blanco:

a) Al procedimiento para resolver un problema en términos de las acciones que se deben ejecutar y del orden en el que se deben ejecutar dichas órdenes se le llama ____________.

b) A la especificación del orden de ejecución de las instrucciones por medio de la computadora se le llama ____________.

c) Todos los programas pueden escribirse en términos de tres tipos de instrucciones de control: ____________, ____________ y ____________.

d) La instrucción de selección ____________ se utiliza para ejecutar una acción cuando una condición es verdadera y otra acción cuando dicha condición es falsa.

e) A muchas instrucciones agrupadas dentro de llaves (**{** y **}**), se les llama ____________.

f) La instrucción de repetición ____________ especifica que una instrucción o grupo de instrucciones se ejecutará de manera repetida mientras una condición sea verdadera.

g) A la repetición de un conjunto de instrucciones, un número específico de veces se le llama repetición ____________.

h) Cuando no se sabe por adelantado el número de veces que se repetirá un conjunto de instrucciones, se puede utilizar un valor ____________ para terminar la repetición.

**3.2** Escriba cuatro instrucciones diferentes de C que sumen 1 a la variable entera **x**.

**3.3** Escriba una instrucción sencilla en C para llevar a cabo cada una de las siguientes tareas:

a) Asigne la suma de **x** y **y** a **z**, e incremente el valor de **x** en 1 después del cálculo.

b) Multiplique la variable **producto** por **2** mediante el uso del operador ***=**.

c) Multiplique la variable **producto** por **2** mediante el uso de los operadores **=** y *****.

d) Verifique si el valor de la variable **cuenta** es mayor que 10. Si lo es, imprima "**Cuenta es mayor que 10**".

e) Decremente la variable **x** en 1, después réstela de la variable **total**.

f) Sume la variable **x** a la variable **total**, después decremente **x** en 1.

g) Calcule el residuo de la división de **q** entre **divisor** y asigne el resultado a **q**. Escriba la instrucción de dos maneras distintas.

h) Imprima el valor **123.4567** con dos dígitos de precisión. ¿Qué valor se imprime?

i) Imprima el valor de punto flotante **3.14159** con tres dígitos de precisión a la derecha del punto decimal. ¿Qué valor se imprime?

**3.4** Escriba una instrucción en C para llevar a cabo cada una de las siguientes tareas:

a) Defina las variables **suma** y **x** de tipo **int**.

b) Inicialice la variable **x** en **1**.

c) Inicialice la variable **suma** en **0**.
d) Sume la variable **x** a la variable **suma** y asigne el resultado a la variable **suma**.
e) Imprima **"La suma es: "** seguida del valor de la variable **suma**.

**3.5** Combine las instrucciones que escribió en el ejercicio 3.4 dentro de un programa que calcule la suma de los enteros 1 a 10. Utilice la instrucción **while** para hacer un ciclo con las instrucciones para el cálculo y el incremento. El ciclo deberá terminar cuando el valor de **x** sea 11.

**3.6** Determine los valores de las variables **producto** y **x** después realizar el cálculo siguiente. Suponga que **producto** y **x** tienen un valor igual que 5 al comenzar la ejecución de la instrucción.

```
producto *= x++;
```

**3.7** Escriba instrucciones sencillas para
a) Introducir la variable entera **x** mediante **scanf**.
b) Introducir la variable entera **y** mediante **scanf**.
c) Inicializar la variable entera **i** en **1**.
d) Inicializar la variable entera **potencia** en **1**.
e) Multiplicar la variable **potencia** por **x** y asignar el resultado a **potencia**.
f) Incrementar la variable **i** en **1**.
g) Verificar **i** para ver si es menor o igual que **y** en la condición de una instrucción **while**.
h) Mostrar la variable entera **potencia** mediante **printf**.

**3.8** Escriba un programa en C que utilice las instrucciones del ejercicio 3.7 para calcular **x** a la potencia **y**. El programa debe tener una instrucción de repetición **while**.

**3.9** Identifique y corrija los errores en cada una de las siguientes instrucciones:
a)
```
while ( c <= 5 ) {
      producto *= c;
      ++c;
```
b)
```
scanf( "%.4f", &valor);
```
c)
```
if ( genero == 1 )
    printf( "Mujer\n" );
else;
    printf( "Hombre\n" );
```

**3.10** ¿Qué es lo que está mal en la siguiente instrucción de repetición **while** (suponga que **z** tiene un valor 100), la cual se supone debe calcular la suma en orden descendente de los enteros de 100 a 1?:
a)
```
while ( z >= 0 )
      suma += z;
```

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**3.1** a) Algoritmo. b) Control de programa. c) Secuencia, selección, repetición. d) **if…else**. e) Instrucción compuesta. f) **while**. g) Controlada por contador. h) Centinela.

**3.2**
```
x = x + 1;
x += 1;
++x;
x++;
```

**3.3**
a)
```
z = x++ + y;
```
b)
```
producto *= 2;
```
c)
```
producto = producto * 2;
```
d)
```
if ( cuenta > 10 )
      printf( "Cuenta es mayor que 10.\n" );
```
e)
```
total -= --x;
```
f)
```
total += x--;
```
g)
```
q %= divisor;
   q = q % divisor;
```
h)
```
printf( "%.2f", 123.4567 );
```
despliega **123.46**.
i)
```
printf( "%.3f\n", 3.14159 );
```
despliega **3.142**.

**3.4** a) **`int suma, x;`**
b) **`x = 1;`**
c) **`suma = 0;`**
d) **`suma += x;`** o **`suma = suma + x;`**
e) **`printf( “La suma es: %d\n”, suma );`**

**3.5** Vea abajo.

```
/* Calcula la suma de los enteros 1 a 10 */
#include <stdio.h>

int main()
{
   int suma, x;        /* define las variables suma y x */

   x = 1;              /* inicializa x */
   suma = 0;           /* inicializa suma */

   while ( x <= 10 ) { /* repite el ciclo mientras x sea menor o igual que */
      suma += x; /* suma x a suma */
      ++x; /* incrementa x */
   } /* fin de while */

   printf( “La suma es: %d\n”, suma ); /* despliega la suma */

   return 0;
} /* fin de la función main */
```

**3.6** **`producto = 25, x = 6;`**

**3.7** a) **`scanf( “%d”, &x );`**
b) **`scanf( “%d”, &y );`**
c) **`i = 1;`**
d) **`potencia = 1;`**
e) **`potencia *= x;`**
f) **`y++;`**
g) **`if ( y <= x )`**
h) **`printf( “%d”, potencia );`**

**3.8** Vea abajo.

```
/* eleva x a la potencia y */
#include <stdio.h>

int main()
{
   int x, y, i, potencia;     /* declaración de las variables */

   i = 1;     /* inicializa i */
   potencia = 1;      /* inicializa potencia */
   scanf( “%d”, &x ); /* lectura de x del usuario */
   scanf( “%d”, &y ); /* lectura de y del usuario */

   while ( i <= y ) { /* repite el ciclo while mientras i sea menor o
   igual que y */
      potencia *= x;  /* multiplica potencia por x */
      ++i;            /* incrementa i */
```

(Parte 1 de 2.)

```
    } /* fin del while */

    printf( "%d", potencia ); /* despliega la potencia */

    return 0;
} /* fin de la función main */
```

(Parte 2 de 2.)

**3.9** a) Error: falta la llave derecha que cierra el cuerpo del **`while`**.
Corrección: añada la llave derecha después de la instrucción **`++c`**.
b) Error: se especifica precisión dentro de la especificación de conversión de **`scanf`**.
Corrección: elimine **`.4`** de la especificación de conversión.
c) Error: el punto y coma después de **`else`** en la instrucción **`if…else`** provoca un error de lógica.
Corrección: elimine el punto y coma después de **`else`**.

**3.10** El valor de la variable **`z`** nunca cambia dentro de la instrucción **`while`**. Por lo tanto, se crea un ciclo infinito. Para prevenir que se presente un ciclo infinito, **`z`** debe disminuir de manera que alcance un valor igual a **`0`**.

## *EJERCICIOS*

**3.11** Identifique y corrija los errores de cada una de las siguientes instrucciones [*Nota:* Puede haber más de un error en cada porción de código]:

a)
```
if ( edad >= 65 );
printf( "La edad es mayor o igual que 65\n" );
else
printf( "La edad es menor que 65\n" );
```
b)
```
int x = 1, total;
while ( x <= 10 ) {
      total += x;
      ++x;
   }
```
c)
```
while ( x <= 100 )
total += x;
++x;
```
d)
```
while ( y > 0 ) {
printf( "%d\n", y );
++y;
    }
```

**3.12** Complete los espacios en blanco:
a) La solución a cualquier problema involucra la realización de una serie de acciones en un ____________ específico.
b) Un sinónimo de procedimiento es ____________.
c) A la variable que acumula la suma de varios números se le llama ____________.
d) Al proceso de asignarles ciertos valores a las variables al principio del programa se le llama ____________.
e) Al valor especial que se utiliza para indicar el "final de la entrada de datos" se le llama ____________, ____________, ____________ o ____________.
f) Un ____________ es una representación gráfica de un algoritmo.
g) En un diagrama de flujo, el orden en el que se deben realizar los pasos se indica mediante símbolos de ____________.
h) El símbolo de terminación indica el ____________ y el ____________ de cada algoritmo.
i) El símbolo rectángulo corresponde a los cálculos que por lo general se realizan por medio de las instrucciones de ____________ y las operaciones de entrada/salida que por lo general se realizan mediante llamadas a ____________ y ____________ de la biblioteca estándar de funciones.
j) Al elemento escrito dentro de un símbolo de decisión se le denomina ____________.

**3.13** ¿Cuál es la salida de la siguiente porción de código?

```
#include <stdio.h>

int main()
{
   int x = 1, total = 0, y;

   while ( x <= 10 ) {
      y = x * x;
      printf( “%d\n”, y );
      total += y;
      ++x;
   }

   printf( “El total es: %d\n”, total );

   return 0;
}
```

**3.14** Escriba una instrucción individual en pseudocódigo que indique cada una de las siguientes acciones:

a) Despliegue el mensaje **“Introduzca dos números”**.

b) Asigne la suma de las variables **x**, **y**, y **z** a la variable **p**.

c) Verifique la siguiente condición dentro de una instrucción de selección **if…else**: el valor actual de la variable **m** es mayor que el doble del valor actual de la variable **v**.

d) Obtenga el valor de las variables **s**, **r**, y **t** desde el teclado.

**3.15** Formule un algoritmo en pseudocódigo para cada una de las siguientes:

a) Obtenga dos números desde el teclado, calcule la suma de los números y despliegue el resultado.

b) Obtenga dos números desde el teclado, y determine y despliegue cuál (si existe) es el mayor de los dos.

c) Obtenga una serie de números positivos desde el teclado, y determine y despliegue la suma de los números. Asuma que el usuario introduce un valor centinela **-1** para indicar el “fin de la entrada de datos”.

**3.16** Indique si las siguientes frases son *verdaderas* o *falsas*. Si una frase es *falsa*, explique por qué.

a) La experiencia ha demostrado que la parte más difícil para solucionar un problema por medio de la computadora es crear un programa funcional en C.

b) Un valor centinela debe ser un valor que no se confunda con un valor de dato legítimo.

c) Las líneas de flujo indican las acciones que se deben realizar.

d) Las condiciones que se escriben dentro de un símbolo de decisión siempre contienen operadores aritméticos (es decir, **+**, **-**, *****, **/**, y **%**).

e) En el mejoramiento arriba-abajo, paso a paso, cada mejora es una representación completa de todo el algoritmo.

***Para los ejercicios 3.17 al 3.21, realice cada uno de los siguientes pasos:***

1. Lea el enunciado del problema.
2. Formule el algoritmo mediante el uso de pseudocódigo y mejoramiento arriba-abajo, paso a paso.
3. Escriba un programa en C.
4. Pruebe, depure y ejecute el programa en C.

**3.17** Los conductores están preocupado por el kilometraje obtenido en sus automóviles. Un conductor mantiene el registro de muchos llenados de tanque de gasolina mediante el registro de miles de kilómetros conducidos y los litros empleados durante cada llenado del tanque. El programa debe calcular y desplegar los kilómetros por litro obtenidos durante cada llenado de tanque. Después de procesar toda la información, el programa debe calcular y desplegar los kilómetros por litro combinados de todos los llenados de tanque. He aquí un ejemplo del diálogo de entrada/salida:

```
Introduzca los litros utilizados (-1 para terminar): 12.8
Introduzca los kilómetros conducidos: 287
Los kilómetros por litro de éste tanque fueron 22.421875
```

```
Introduzca los litros utilizados (-1 para terminar): 10.3
Introduzca los kilómetros conducidos: 200
Los kilómetros por litro de éste tanque fueron 19.417475

Introduzca los litros utilizados (-1 para terminar): 5
Introduzca los kilómetros conducidos: 120
Los kilómetros por litro de éste tanque fueron 24.000000

Introduzca los litros utilizados (-1 para terminar): -1

El promedio general de kilómetros/litro fue 21.601423
```

**3.18** Desarrolle un programa en C que determine si un cliente de una tienda departamental excede el límite de crédito de su cuenta. Para cada cliente, se dispone de los siguientes datos:

1. Número de cuenta.
2. Saldo al inicio del mes.
3. Total de elementos cargados al cliente en este mes.
4. El total de los créditos aplicados a la cuenta del cliente durante el mes.
5. El límite de crédito autorizado.

El programa debe introducir cada uno de estos datos, calcular el nuevo saldo (= *saldo inicial* + *cargos* – *créditos*), y determinar si el nuevo saldo excede el límite de crédito del cliente. Para aquellos clientes que excedan el límite de crédito, el programa debe desplegar el número de cuenta, el límite de crédito, el saldo nuevo y el mensaje "Límite de crédito excedido". A continuación se muestra un ejemplo del diálogo de entrada/salida:

```
Introduzca el número de cuenta (-1 para terminar): 100
Introduzca el saldo inicial: 5394.78
Introduzca el total de cargos: 1000.00
Introduzca el total de créditos: 500.00
Introduzca el límite de crédito: 5500.00
Cuenta: 100
Límite de crédito: 5500.00
Saldo: 5894.78
Límite de crédito excedido.

Introduzca el número de cuenta (-1 para terminar): 200
Introduzca el saldo inicial: 1000.00
Introduzca el total de cargos: 123.45
Introduzca el total de créditos: 321.00
Introduzca el límite de crédito: 1500.00

Introduzca el número de cuenta (-1 para terminar): 300
Introduzca el saldo inicial: 500.00
Introduzca el total de cargos: 274.73
Introduzca el total de créditos: 100.00
Introduzca el límite de crédito: 800.00

Introduzca el número de cuenta (-1 para terminar): -1
```

**3.19** Una gran empresa de productos químicos le paga a sus vendedores mediante un esquema de comisiones. Los vendedores reciben $200 semanales más el 9% de sus ventas totales durante la semana. Por ejemplo, un vendedor que vende $5000 de productos químicos durante la semana recibe $200 más el 9% de $5000, o un total de $650. Desarrolle un programa que introduzca las ventas totales de cada vendedor durante la última semana y que calcule y despliegue los ingresos de ese vendedor. Procese las cantidades de un vendedor a la vez. A continuación se muestra un ejemplo del diálogo de entrada/salida:

```
Introduzca las ventas en pesos (-1 para terminar): 5000.00
El salario es: $650.00

Introduzca las ventas en pesos (-1 para terminar): 1234.56
El salario es: $311.11

Introduzca las ventas en pesos (-1 para terminar): 1088.89
El salario es: $298.00

Introduzca las ventas en pesos (-1 para terminar): -1
```

**3.20** El interés simple para un préstamo se calcula mediante la fórmula:

**`interés = préstamo * tasa * días / 365;`**

La fórmula anterior asume que **`tasa`** es la tasa de interés anual, y por lo tanto incluye la división entre 365 (días). Desarrolle un programa que introduzca **`préstamo`**, **`tasa`** y **`días`** para varios préstamos, y que calcule y despliegue el interés simple para cada préstamo, utilizando la fórmula anterior. A continuación se muestra un ejemplo del diálogo de entrada/salida:

```
Introduzca el monto del préstamo (-1 para terminar): 1000.00
Introduzca la tasa de interés: .1
Introduzca el periodo del préstamo en días: 365
El monto del interés es $100.00

Introduzca el monto del préstamo (-1 para terminar): 1000.00
Introduzca la tasa de interés: .08375
Introduzca el periodo del préstamo en días: 224
El monto del interés es $51.40

Introduzca el monto del préstamo (-1 para terminar): 10000.00
Introduzca la tasa de interés: .09
Introduzca el periodo del préstamo en días: 1460
El monto del interés es $3600.00

Introduzca el monto del préstamo (-1 para terminar): -1
```

**3.21** Desarrolle un programa que determine el pago bruto de cada uno de los empleados. Esta empresa paga "horas completas" por las primeras 40 horas trabajadas por cada empleado y paga "hora y media" por todas las horas extras trabajadas después de las 40. Usted tiene una lista de los empleados de la empresa, el número de horas que trabajó cada empleado la semana pasada y el pago por hora de cada empleado. Su programa deberá introducir esta información para cada empleado, y deberá determinar y desplegar el pago bruto por empleado. A continuación, mostramos un ejemplo del diálogo de entrada/salida:

```
Introduzca el No. de horas laboradas (-1 para terminar): 39
Introduzca el pago por hora del empleado: 10.00
El salario es: $390.00

Introduzca el No. de horas laboradas (-1 para terminar): 40
Introduzca el pago por hora del empleado: 10.00
El salario es: $400.00

Introduzca el No. de horas laboradas (-1 para terminar): 41
Introduzca el pago por hora del empleado: 10.00
El salario es: $415.00

Introduzca el No. de horas laboradas (-1 para terminar): -1
```

**3.22** Escriba un programa que demuestre la diferencia entre el predecremento y el posdecremento mediante el uso del operador `--`.

**3.23** Escriba un programa que utilice un ciclo para imprimir los números 1 a 10 dentro de la misma línea, separados cada uno por tres espacios en blanco.

**3.24** El proceso para encontrar el número más grande (es decir, el máximo de un grupo de números) se utiliza con frecuencia en aplicaciones para computadora. Por ejemplo, un programa que determina el ganador de un concurso de unidades vendidas por cada vendedor. El vendedor que vende el mayor número de unidades gana. Escriba un programa en pseudocódigo y posteriormente un programa que introduzca una serie de 10 números y determine e imprima el mayor de éstos. [*Clave:* Su programa debe utilizar tres variables de la siguiente manera]:

**`contador:`** Un contador para contar los números de 1 a 10 (es decir, para llevar la cuenta de cuántos números se han introducido y determinar si ya se procesaron los 10 números).
**`numero:`** El número actual que se introduce al programa.
**`mayor:`** El número más grande encontrado hasta el momento.

**3.25** Escriba un programa que utilice ciclos para imprimir la siguiente tabla de valores.

```
N          10*N        100*N      1000*N

1          10          100         1000
2          20          200         2000
3          30          300         3000
4          40          400         4000
5          50          500         5000
6          60          600         6000
7          70          700         7000
8          80          800         8000
9          90          900         9000
10         100         1000        10000
```

La secuencia de escape tabulador, **`\t`**, puede utilizarse en la instrucción **`printf`** para separar las columnas con tabuladores.

**3.26** Escriba un programa que utilice ciclos para producir la siguiente tabla de valores:

```
A          A+2         A+4        A+6

3          5           7           9
6          8           10          12
9          11          13          15
12         14          16          18
15         17          19          21
```

**3.27** Mediante un método similar al del ejercicio 3.24, encuentre los *dos* valores más grandes de los 10 números. [*Nota:* Debe introducir un número a la vez.]

**3.28** Modifique el programa de la figura 3.10 para validar sus entradas. Para cualquier entrada, si el valor introducido es diferente a 1 o 2, continúe el ciclo hasta que el usuario digite un valor correcto.

**3.29** ¿Qué despliega el siguiente programa?

```
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   int contador = 1; /* inicializa contador */

   while ( contador <= 10 ) { /* repite 10 veces */
```

(Parte 1 de 2.)

```

      /* muestra una línea de texto */
      printf( "%s\n", contador % 2 ? "****" : "++++++++" );
      contador++; /* incrementa contador */
   } /* fin de while */

   return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

(Parte 2 de 2.)

**3.30** ¿Qué despliega el siguiente programa?

```
#include <stdio.h>

/* la función main inicia la ejecución del programa */
int main()
{
   int fila = 10; /* inicializa la fila */
   int columna;   /* declara columna */

   while ( fila >= 1 ) { /* repite el ciclo hasta que fila < 1 */
      columna = 1;       /* establece la columna en 1 al comenzar la
      iteración */

      while ( columna <= 10 ) {                 /* repite 10 veces */
         printf( "%s", fila % 2 ? "<": ">" );   /* salida */
         columna++;                             /* incrementa columna */
     } /* fin del while interno */

      fila--;         /* decrementa fila */
      printf( "\n" ); /* comienza la nueva línea de salida */
   } /* fin del while externo */

   return 0; /* indica que el programa terminó con éxito */

} /* fin de la función main */
```

**3.31** (*Problema de asociación de else.*) Determine la salida para cada una de las siguientes variables, cuando **x** es **9** y **y** es **11**, y cuando **x** es **11** y **y** es **9**. Observe que el compilador ignora el sangrado de un programa en C. Además, el compilador siempre asocia un **else** con su **if** previo, a menos que se le indique lo contrario mediante la colocación de llaves **{}**. Debido, en primera instancia, a que el programador puede no estar seguro cuál es el **if** que coincide con el **else**, a este problema se le conoce como el problema de "asociación de else". Eliminamos el sangrado del siguiente código para hacer el problema más interesante. [*Pista:* Aplique las convenciones de sangrado que aprendió.]

a)
```
if ( x < 10 )
if ( y > 10 )
printf( "*****\n" );
else
printf( "#####\n" );
printf( "$$$$$\n" );
```

b)
```
if ( x < 10 ) {
if ( y > 10 )
printf( "*****\n" );
}
else {
```

```
printf( "#####\n" );
printf( "$$$$$\n" );
}
```

**3.32** (*Otro problema de asociación de else.*) Modifique el siguiente código para producir la salida que aparece a continuación. Utilice las técnicas de sangrado apropiadas. No debe hacer cambio alguno que no sea el de insertar llaves. El compilador ignora el sangrado de un programa. Eliminamos el sangrado del siguiente código para hacer más interesante el problema. [*Nota:* Es posible que no sea necesario hacer modificaciones.]

```
if ( y == 8 )
if ( x == 5 )
printf( "@@@@@\n" );
else
printf( "#####\n" );
printf( "$$$$$\n" );
printf( "&&&&&\n" );
```

a) Si suponemos que **x = 5** y **y = 8**, se produce la siguiente salida.

```
@@@@@
$$$$$
&&&&&
```

b) Si suponemos que **x = 5** y **y = 8**, se produce la siguiente salida.

```
@@@@@
```

c) Si suponemos que **x = 5** y **y = 8**, se produce la siguiente salida.

```
@@@@@
&&&&&
```

d) Si suponemos que **x = 5** y **y = 7**, se produce la siguiente salida. [*Nota:* Las últimas tres instrucciones **printf** son parte de una instrucción compuesta.]

```
#####
$$$$$
&&&&&
```

**3.33** Escriba un programa que lea la medida de uno de los lados de un cuadrado y que despliegue dicho cuadrado con asteriscos. Su programa debe trabajar con cuadrados de tamaño entre 1 y 20. Por ejemplo, si su programa lee un tamaño 4, debe desplegar:

```
****
****
****
****
```

**3.34** Modifique el programa que escribió en el ejercicio 3.33 de manera que despliegue el perímetro del cuadrado. Por ejemplo, si su programa lee un tamaño 5, debe desplegar:

```
*****
*   *
*   *
*   *
*****
```

**3.35** Un palíndromo es un número o una frase de texto que se lee igual hacia delante y hacia atrás. Por ejemplo, cada uno de los siguientes números de cinco dígitos, son palíndromos: 12321, 55555, 45554, y 11611. Escriba un programa que lea números de cinco dígitos y que determine si es o no, un palíndromo. [*Pista:* Utilice los operadores de división y residuo para separar el número en sus dígitos individuales.]

**3.36** Introduzca un número entero que contenga sólo unos y ceros (es decir, un entero "binario") y que despliegue su equivalente decimal. [*Pista:* Utilice los operadores de división y residuo para separar los dígitos del número "binario", uno por uno, de derecha a izquierda. Así como en el sistema de numeración decimal, el dígito más a la derecha tiene un valor de posición de 1, y el siguiente dígito a la izquierda tiene un valor por posición de 10, después de 100, después de 1000, y así sucesivamente, en el sistema binario de numeración, el dígito que se encuentra a la derecha tiene un valor por posición de 1, el siguiente dígito a la izquierda tiene un valor por posición de 2, después de 4, de 8, y así sucesivamente. Así, el número 234 se puede interpretar como 4 * 1 + 3 * 10 + 2 * 100. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 o 1 + 0 + 4 + 8 o 13.]

**3.37** ¿Cómo puede determinar la rapidez real con la que opera su propia computadora? Escriba un programa mediante un ciclo **`while`** que cuente de 1 a 300,000,000 por unos. Cada vez que la cuenta alcance un múltiplo de 100,000,000 despliegue dicho número en la pantalla. Utilice su reloj para determinar cuánto tarda cada millón de repeticiones del ciclo.

**3.38** Escriba un programa que despliegue 100 asteriscos, uno a la vez. Después de cada diez asteriscos, el programa debe desplegar un carácter de nueva línea. [*Pista:* Cuente de 1 a 100. Utilice el operador módulo para reconocer cada vez que el contador alcance un múltiplo de 10.]

**3.39** Escriba un programa que lea un número entero y que determine y despliegue cuántos dígitos del entero son sietes.

**3.40** Escriba un programa que despliegue el siguiente patrón en la pantalla:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

El programa sólo debe utilizar tres instrucciones de salida, una de cada una de las siguientes formas:

```
printf( "* ");
printf( " ");
printf( "\n");
```

**3.41** Escriba un programa que despliegue los múltiplos del número entero 2, a saber 2, 4, 8, 16, 32, 64, y así sucesivamente. Su ciclo no debe terminar (es decir, debe crear un ciclo infinito). ¿Qué sucede cuando ejecuta este programa?

**3.42** Escriba un programa que lea el radio de un círculo (como un valor **`float`**) y que calcule y despliegue el diámetro, la circunferencia y el área. Utilice el valor 3.14159 para π.

**3.43** ¿Qué está mal en la siguiente instrucción? Rescriba la instrucción para realizar lo que probablemente intentaba hacer el programador.

```
printf( "%d", ++( x + y ) );
```

**3.44** Escriba un programa que lea tres valores de tipo **`float`** diferentes de cero y que determine (y despliegue) si éstos pueden representar los lados de un triángulo recto.

**3.45** Escriba un programa que lea tres enteros diferentes de cero y que determine (y despliegue) si pueden representar los lados de un triángulo recto.

**3.46** Una empresa quiere transmitir datos mediante la línea telefónica, pero les preocupa que sus teléfonos pudieran estar intervenidos. Todos sus datos se transmiten como enteros de cuatro dígitos. A usted le pidieron que escriba un programa que encripte sus datos de manera que se transmitan de forma más segura. El programa debe leer un entero de cuatro dígitos y encriptar la información de la siguiente manera: reemplace cada dígito con el residuo de la división entre *10* de *la suma de dicho dígito más 7*. Posteriormente, intercambie el primer dígito con el tercero, e intercambie el segundo dígito con el cuarto. Luego despliegue el entero encriptado. Escriba un programa por separado que introduzca un entero encriptado de cuatro dígitos y lo desencripte para formar el número original.

**3.47** El factorial de un número entero positivo $n$ se escribe $n!$ (que se pronuncia "$n$ factorial") y se define como:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (para valores de $n$ mayores o iguales que 1)

y

$n! = 1$ (para $n = 0$)

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

a) Escriba un programa que lea un entero positivo y que calcule y despliegue su factorial.

b) Escriba un programa que estime el valor de la constante matemática $e$, mediante el uso de la fórmula:

$$e = 1 + \frac{1}{\ } + \frac{1}{\ } + \frac{1}{\ } + \ldots$$

c) Escriba un programa que calcule el valor de $e^x$ mediante el uso de la fórmula:

$$e^x = 1 + \frac{x}{\ } + \frac{x^2}{\ } + \frac{x^3}{\ } + \ldots$$



# Control de programas en C

## Objetivos

- Utilizar las instrucciones de repetición **`for`** y **`do…while`**.
- Comprender la selección múltiple a través de la instrucción de selección **`switch`**.
- Utilizar las instrucciones de control de programa **`break`** y **`continue`**.
- Utilizar los operadores lógicos.




*¿Quién puede controlar su destino?*
William Shakespeare
*Otelo*

*La llave utilizada siempre es brillante.*
Benjamin Franklin

*El hombre es un animal generador de herramientas.*
Benjamin Franklin

*La inteligencia… es la capacidad de crear objetos artificiales, en especial, herramientas para hacer herramientas.*
Henry Bergson

## Plan general



## 4.1 Introducción

Ahora, el lector debe sentirse cómodo con el proceso de escribir programas sencillos, pero completos, en C. En este capítulo, trataremos con mucho detalle la repetición, y también presentaremos otras instrucciones de control de repetición, a saber, las instrucciones **`for`** y **`do...while`**; así como la instrucción de selección múltiple **`swith`**. Explicaremos la instrucción **`break`**, para salir rápidamente de ciertas instrucciones de control, y la instrucción **`continue`** para saltar el resto del cuerpo de una instrucción de repetición y continuar con la siguiente iteración del ciclo. El capítulo explica los operadores lógicos utilizados para combinar condiciones, y concluye con un resumen sobre los principios de la programación estructurada que presentamos en los capítulos 3 y 4.

## 4.2 Fundamentos de la repetición

La mayoría de los programas involucran la repetición, o *ciclos*. Un *ciclo* es un grupo de instrucciones que la computadora ejecuta repetidamente, mientras alguna *condición de continuación de ciclo* permanezca verdadera. Hemos explicado dos medios para llevar a cabo una repetición:

1. Repetición controlada por contador.
2. Repetición controlada por centinela.

En algunas ocasiones, a la repetición controlada por contador se le conoce como *repetición definida*, ya que sabemos por adelantado el número exacto de veces que se ejecutará el ciclo; y a la repetición controlada por centinela a veces se le llama *repetición indefinida*, ya que no sabemos por adelantado cuántas veces se ejecutará el ciclo.

En la repetición controlada por contador, se utiliza una *variable de control* para contar el número de repeticiones. La variable de control se incrementa (por lo general en 1) cada vez que el grupo de instrucciones se ejecuta. Cuando el valor de la variable de control indica que el número correcto de repeticiones se ha alcanzado, el ciclo termina y la computadora continúa con la ejecución de la instrucción que se encuentra después de la instrucción de repetición.

Se utilizan valores centinela para controlar una repetición cuando:

1. No conocemos por adelantado el número preciso de repeticiones.
2. El ciclo incluye instrucciones que obtienen datos, cada vez que el ciclo se ejecuta.

El valor centinela indica “fin de los datos”. El centinela se introduce después de que se le proporcionaron al programa todos los datos regulares. Los centinelas deben ser diferentes de los elementos de datos regulares.

## 4.3 Repetición controlada por contador

La repetición controlada por contador requiere:

1. El *nombre* de una variable de control (o contador de ciclo).
2. El *valor inicial* de la variable de control.
3. El *incremento* (o *decremento*) mediante el cual se modifica la variable de control cada vez que se repite el ciclo.
4. La condición que evalúa el *valor final* de la variable de control (es decir, si el ciclo debe continuar).

Considere el sencillo programa de la figura 4.1, el cual despliega los números del 1 al 10. La declaración

```
int contador = 1; /* inicialización */
```

*nombra* a la variable de control (**contador**), la declara como entero, reserva espacio en memoria para ella, y le asigna un *valor inicial* de **1**. Esta declaración no es una instrucción ejecutable.

La declaración e inicialización de **contador** pudo haberse hecho con las instrucciones

```
int contador;
contador = 1;
```

La declaración no es ejecutable, pero la asignación sí lo es. Nosotros utilizamos ambos métodos para inicializar variables.

```
/* Figura 4.1: fig04_01.c
   Repetición controlada por contador */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int contador = 1; /* inicialización */

   while ( contador <= 10 ) { /* condición de repetición */
      printf ( “%d\n”, contador ); /* despliega el contador */
      ++contador; /* incremento */
   } /* fin del while */

   return 0; /* indica terminación exitosa */

} /* fin de la función main */
```

```
1
2
3
4
5
6
7
8
9
10
```

**Figura 4.1** Repetición controlada por contador.

La instrucción

```
++contador; /* incremento */
```

*incrementa* en 1 al contador del ciclo, cada vez que éste se ejecuta. La condición de continuación de ciclo correspondiente a la instrucción **`while`** evalúa si el valor de la variable de control es menor o igual que **`10`** (el último valor con el que la condición es verdadera). Observe que el cuerpo de este **`while`** se ejecuta incluso si la variable de control es **`10`**. El ciclo termina cuando la variable de control excede a **`10`** (es decir, cuando **`contador`** toma el valor de **`11`**).

Los programadores en C normalmente harían más conciso el programa de la figura 4.1, inicializando **`contador`** en **`0`**, y reemplazando la instrucción **`while`** con

```
while ( ++contador <= 10 )
   printf( "%d\n", contador );
```

Este código nos ahorra una instrucción, ya que el incremento se hace directamente en la condición **`while`**, antes de que se evalúe la condición. Además, este código elimina la necesidad de llaves alrededor del cuerpo de **`while`**, ya que éste ahora contiene sólo una instrucción. Escribir código de manera condensada requiere cierta práctica.



**Error común de programación 4.1**

*Debido a que los valores de punto flotante pueden ser aproximados, controlar ciclos contadores con variables de punto flotante puede dar como resultado valores contadores imprecisos y evaluaciones de terminación incorrectas.*



**Tip para prevenir errores 4.1**

*Controle los ciclos contadores con valores enteros.*



**Buena práctica de programación 4.1**

*Sangre las instrucciones correspondientes al cuerpo de toda instrucción de control.*



**Buena práctica de programación 4.2**

*Coloque una línea en blanco antes y después de cada instrucción de control, para que resalten en el programa.*



**Buena práctica de programación 4.3**

*Tener demasiados niveles de anidamiento, puede provocar que un programa sea difícil de entender. Como regla general, intente evitar el uso de más de tres niveles de anidamiento.*



**Buena práctica de programación 4.4**

*Combinar espaciado vertical, antes y después de las instrucciones de control, con sangría en los cuerpos de dichas instrucciones, proporciona a los programas una apariencia bidimensional, la cual mejora bastante la legibilidad del programa.*

## 4.4 Instrucción de repetición `for`

La instrucción de repetición **`for`** maneja todos los detalles de la repetición controlada por contador. Para ilustrar el poder de **`for`**, rescribamos el programa de la figura 4.1. El resultado aparece en la figura 4.2.

```
/* Figura 4.2: fig04_02.c
   Repetición controlada por contador mediante la instrucción for */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()

   int contador; /* definición del contador */
```

**Figura 4.2** Repetición controlada por contador mediante la instrucción **`for`**. (Parte 1 de 2.)

```
 9
10      /* la inicialización, condición de repetición, e incremento
11         se incluyen en el encabezado de la instrucción for */
12      for ( contador = 1; contador <= 10; contador++ ) {
13         printf( "%d\n", contador );
14       } /* fin del for */
15
16      return 0; /* indica terminación exitosa del programa */
17
18   } /* fin de la función main */
```

**Figura 4.2** Repetición controlada por contador mediante la instrucción **`for`**. (Parte 2 de 2.)

El programa funciona de la siguiente manera. Cuando la instrucción **`for`** comienza a ejecutarse, la variable de control **`contador`** se inicializa en **`1`**. Después, se evalúa la condición de continuación de ciclo, **`contador <= 10`**. Debido a que el valor inicial de **`contador`** es **`1`**, la condición se satisface, por lo que la instrucción **`printf`** (línea 13) imprime el valor de **`contador`**, es decir, **`1`**. En seguida, la variable de control **`contador`** se incrementa por medio de la expresión **`contador++`**, y el ciclo comienza nuevamente con la evaluación de la condición de continuación de ciclo. Ya que ahora la variable de control es igual a **`2`**, el valor final no es excedido, por lo que el programa ejecuta nuevamente la instrucción **`printf`**. Este proceso continúa hasta que la variable de control, **`contador`**, se incrementa a su valor final 11; esto ocasiona que la evaluación de la condición de continuación de ciclo falle, y la repetición termina. El programa continúa con la ejecución de la primera instrucción posterior a **`for`** (en este caso, la instrucción **`return`** que se encuentra al final del programa).

La figura 4.3 echa un vistazo más cercano a la instrucción **`for`** de la figura 4.2. Observe que dicha instrucción "lo hace todo"; especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si hay más de una instrucción en el cuerpo de **`for`**, es necesario utilizar llaves para definir el cuerpo del ciclo.

Observe que la figura 4.2 utiliza la condición de continuación de ciclo, **`contador <= 10`**. Si el programador escribió incorrectamente **`contador < 10`**, entonces el ciclo se ejecutaría sólo 9 veces. Éste es un error común de lógica llamado *error de desplazamiento en uno*.



### Error común de programación 4.2

*Utilizar un operador de relación incorrecto o usar un valor final incorrecto en un contador de ciclo, dentro de la condición de una instrucción* **`while`** *o* **`for`**, *puede ocasionar errores por desplazamiento en uno.*



### Tip para prevenir errores 4.2

*Utilizar el valor final en la condición de una instrucción* **`while`** *o* **`for`**, *y utilizar el operador de relación* **`<=`**, *ayudará a evitar errores por desplazamiento en uno. Por ejemplo, para un ciclo utilizado para imprimir los valores del 1 al 10, la condición de continuación de ciclo debe ser* **`contador <= 10`**, *en lugar de* **`contador < 11`** *o* **`contador < 10`**.

El formato general de la instrucción **`for`** es:

> **`for`** ( *expresión1*; *expresión2*; *expresión3* )
> *instrucción*

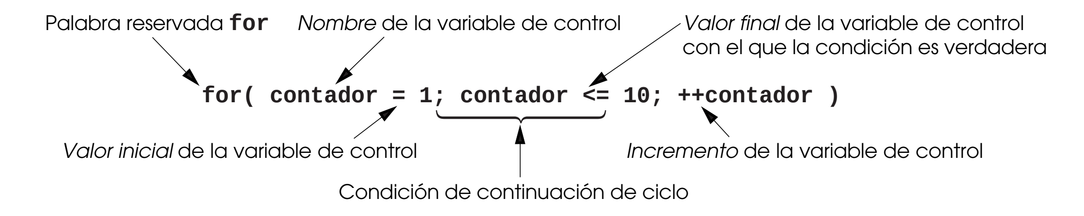



**Figura 4.3** Componentes del encabezado **`for`**.

en donde *expresión1* inicializa la variable de control de ciclo, *expresión2* es la condición de continuación de ciclo, y *expresión3* incrementa la variable de control. En la mayoría de los casos, la instrucción **`for`** puede representarse con una instrucción **`while`** equivalente, de la siguiente manera:

```
expresión1;
while ( expresión2 ) {
    instrucción
    expresión3;
}
```

Existe una excepción a esta regla, la cual explicaremos en la sección 4.9.

Con frecuencia, *expresión1* y *expresión3* son listas de expresiones separadas por comas. Las comas, como las usamos aquí, son en realidad *operadores coma* que garantizan que esas listas de expresiones se evalúen de izquierda a derecha. El valor y el tipo de una lista de expresiones separadas por comas es el valor y el tipo de la expresión que se encuentra más a la derecha de la lista. El operador coma se utiliza con mucha frecuencia en instrucciones **`for`**. Su principal aplicación es la de permitir al programador que utilice múltiples expresiones de inicialización y/o múltiples expresiones de incremento. Por ejemplo, puede haber distintas variables de control en una sola instrucción **`for`** que deben inicializarse e incrementarse.

### Observación de ingeniería de software 4.1

*Dentro de las secciones de inicialización e incremento de una instrucción **`for`**, sólo coloque expresiones relacionadas con las variables de control. La manipulación de otro tipo de variables debe aparecer ya sea antes del ciclo (si se deben ejecutar sólo una vez, como las instrucciones de inicialización), o dentro del cuerpo del ciclo (si se deben ejecutar una vez por repetición, como las instrucciones de incremento y decremento).*

Las tres expresiones de la instrucción **`for`** son opcionales. Si se omite la *expresión2*, C asume que la condición es verdadera, con lo que se genera un ciclo infinito. Es posible omitir la *expresión1*, si la variable de control se inicializa en alguna otra parte del programa. La *expresión3* podría omitirse, si el incremento lo calculan las expresiones del cuerpo de la instrucción **`for`**, o si no se necesita incremento alguno. La expresión de incremento correspondiente a la instrucción **`for`** actúa como una instrucción de C independiente al final del cuerpo de **`for`**. Por lo tanto, las expresiones:

```
contador = contador + 1
contador += 1
++contador
contador++
```

son equivalentes como incremento en la instrucción **`for`**. Muchos programadores en C prefieren **`contador++`**, ya que el incremento ocurre después de que se ejecuta el cuerpo del ciclo, y la forma de postincremento luce más natural. Debido a que la variable que aquí se preincrementa o se postincrementa no aparece en una expresión, ambas formas de incremento tienen el mismo efecto. Los dos puntos y coma de la instrucción **`for`** son necesarios.



### Error común de programación 4.3

*Utilizar comas en lugar de puntos y comas en un encabezado **`for`**, es un error de sintaxis.*



### Error común de programación 4.4

*Colocar un punto y coma inmediatamente a la derecha del paréntesis de un encabezado **`for`**, convierte el cuerpo de dicha instrucción en una instrucción vacía. Por lo general, éste es un error lógico.*

## 4.5 Instrucción `for`: Notas y observaciones

1. La inicialización, la condición de continuación de ciclo y el incremento pueden contener expresiones aritméticas. Por ejemplo, si **`x = 2`** y **`y = 10`**, la instrucción:

```
for ( j = x; j <= 4 * x * y; j +0 y / x )
```

es equivalente a la instrucción:

```
for ( j = 2; j <= 80; j += 5 )
```

2. El "incremento" puede ser negativo (en cuyo caso, en realidad sería un decremento, y el ciclo en realidad contaría hacia atrás).
3. Si de inicio, la condición de continuación de ciclo es falsa, la parte del cuerpo del ciclo no se ejecuta. En su lugar, la ejecución procede con la instrucción que sigue a la instrucción **`for`**.
4. La variable de control con frecuencia se imprime o se utiliza en cálculos realizados en el cuerpo del ciclo, pero no tiene que ser así. Es común utilizar la variable de control para controlar la repetición, sin tener que mencionarla en el cuerpo del ciclo.
5. El diagrama de flujo de una instrucción **`for`** es muy parecido al de la instrucción **`while`**. Por ejemplo, el diagrama de flujo de la instrucción **`for`**

```
for ( contador = 1; contador <= 10; contador++ )
    printf( "%d", contador );
```

aparece en la figura 4.4. Este diagrama de flujo pone en claro que la inicialización ocurre sólo una vez, y que el incremento ocurre después de que la instrucción del cuerpo se ejecuta. Observe que (además de pequeños círculos y flechas) el diagrama de flujo contiene sólo símbolos rectángulos y rombos. De nuevo, imagine que el programador tiene acceso a un gran montón de instrucciones **`for`** vacías (representadas como segmentos del diagrama de flujo), tantas como necesite apilar y anidar con otras instrucciones de control, para formar una implementación estructurada del flujo de control de un algoritmo. Y, de nuevo, los rectángulos y los rombos se llenan con acciones y decisiones apropiadas para el algoritmo.



**Tip para prevenir errores 4.3**

*Aunque el valor de la variable de control puede modificarse en el cuerpo de un ciclo* **`for`**, *esto puede provocar errores sutiles. Es mejor no cambiarlo.*

## 4.6 Ejemplos de la utilización de la estructura `for`

Los siguientes ejemplos muestran los métodos para modificar la variable de control en una instrucción **`for`**.

1. Modifique la variable de control de **`1`** a **`100`**, en incrementos de **`1`**.

```
for ( i = 1; i <= 100; i++ )
```

2. Modifique la variable de control de **`100`** a **`1`** en incrementos de **`-1`** (decrementos de **`1`**).

```
for (i = 100; i >= 1; i-- )
```

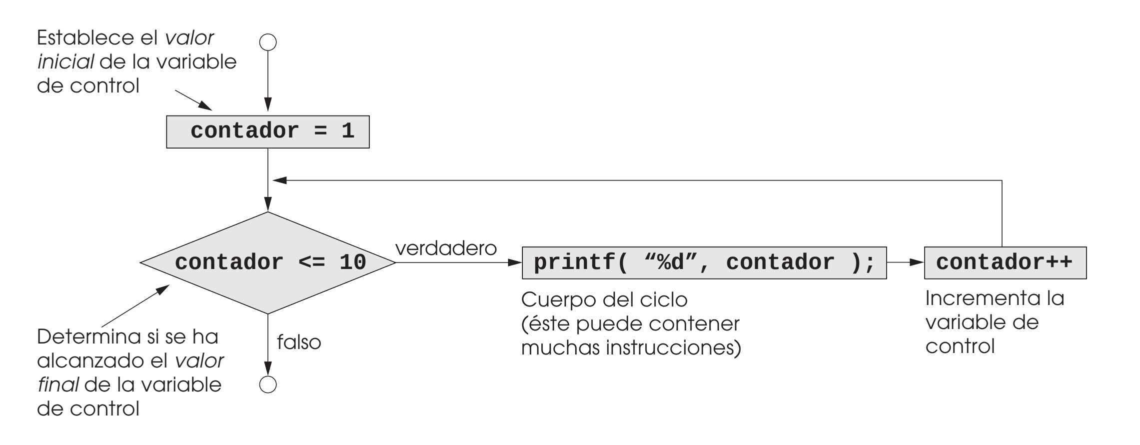



**Figura 4.4** Diagrama de flujo de una instrucción típica de repetición **`for`**.

3. Modifique la variable de control de **7** a **77** en pasos de **7**.

```
for ( i = 7; i <= 77; i += 7 )
```

4. Modifique la variable de control de **20** a **2** en pasos de **-2**.

```
for ( i = 20; i >= 2; i -= 2 )
```

5. Modifique la variable de control en la siguiente secuencia de valores: **2**, **5**, **8**, **11**, **14**, **17**, **20**.

```
for ( j = 2; j <= 20; j += 3 )
```

6. Modifique la variable de control en la siguiente secuencia de valores: **99**, **88**, **77**, **66**, **55**, **44**, **33**, **22**, **11**, **0**.

```
for ( j = 99; j >= 0; j -= 11 )
```

Los dos siguientes ejemplos proporcionan aplicaciones sencillas para la instrucción **for**. La figura 4.5 utiliza la instrucción **for** para sumar todos los enteros pares del **2** al **100**.

Observe que el cuerpo de la instrucción **for** de la figura 4.5 se pudo haber fusionado dentro de la parte que se encuentra más a la derecha del encabezado de **for**, mediante el operador coma, de la siguiente forma:

```
for ( numero = 2; numero <= 100; suma += numero, numero += 2 )
   ; /* instrucción vacía */
```

La inicialización **suma = 0** también pudo haberse fusionado en la sección de inicialización del **for**.



### Buena práctica de programación 4.5

*Aunque las instrucciones que preceden a* **for** *y las instrucciones del cuerpo de un* **for***, a menudo se pueden fusionar dentro de un encabezado* **for***, evite hacerlo, ya que esto ocasiona que el programa sea más difícil de leer.*



### Buena práctica de programación 4.6

*Si es posible, limite el tamaño de los encabezados de las instrucciones de control a una sola línea.*

```
/* Figura 4.5: fig04_05.c
   Suma con for */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int suma = 0; /* inicializa la suma */
   int numero;   /* número por adicionar a suma */

   for ( numero = 2; numero <= 100; numero += 2 ) {
      suma += numero; /* suma el número a suma */
   } /* fin de for */

   printf( "La suma es %d\n", suma ); /* muestra la suma */

   return 0; /* indica terminación exitosa */

} /* fin de la función main */
```

```
La suma es 2550
```

**Figura 4.5** Uso de la instrucción **for** para sumar números.

El siguiente ejemplo calcula el interés compuesto, utilizando la instrucción **for**. Considere el siguiente enunciado del problema:

*Una persona invierte $1000.00 en una cuenta de ahorros con un 5% de interés. Se asume que todo el interés se deja en depósito dentro de la cuenta; calcule y despliegue el monto acumulado de la cuenta al final de cada año, durante 10 años. Utilice la siguiente fórmula para determinar estos montos:*

$$a = p(1 + r)^n$$

*donde*

- $p$ *es el monto de la inversión original (es decir, la inversión principal)*
- $r$ *es la tasa de interés anual*
- $n$ *es el número de años*
- $a$ *es el monto del depósito al final del año n.*

Este problema involucra un ciclo que realiza el cálculo indicado para cada uno de los 10 años en los que el dinero permanece en depósito. La solución aparece en la figura 4.6. [*Nota:* En muchos compiladores de C UNIX, usted debe incluir la opción **-lm** (por ejemplo, **cc -lm fig04_06.c**), cuando compile el programa de la figura 4.6. Esto vincula a la biblioteca de funciones matemáticas con el programa.]

La instrucción **for** ejecuta 10 veces el cuerpo del ciclo, modificando una variable de control del 1 al 10, en incrementos de 1. Aunque C no incluye un operador de exponenciación, para este propósito podemos utilizar la función **pow** de la biblioteca estándar. La función **pow(x, y)** calcula el valor de **x** elevado a la potencia **y**. Ésta toma dos argumentos de tipo ***double*** y devuelve un valor del mismo tipo. El tipo **double** es un tipo de punto flotante muy parecido a **float**, pero en general, una variable de tipo **double** puede almacenar un valor mucho más grande con una mayor precisión que **float**. Observe que siempre que se utilice una función como **pow**, debe incluirse el encabezado **math.h** (línea 4). De hecho, este programa no funcionaría bien si no incluyéramos

```
1  /* Figura 4.6: fig04_06.c
2     Cálculo del interés compuesto */
3  #include <stdio.h>
4  #include <math.h>
5
6  /* la función main comienza la ejecución del programa */
7  int main()
8  {
9     double monto;               /* monto del depósito */
10    double principal = 1000.0;  /* monto principal */
11    double tasa = .05;          /* interés compuesto anual */
12    int anio;                   /* contador de años */
13
14    /* muestra el encabezado de salida de la tabla */
15    printf( "%4s%21s\n", "Anio", "Monto del deposito" );
16
17    /* calcula el monto del depósito para cada uno de los diez años */
18    for ( anio = 1; anio <= 10; anio++ ) {
19
20       /* calcula el nuevo monto para el año especificado */
21       monto = principal * pow( 1.0 + tasa, anio );
22
23       /* muestra una línea de la tabla */
24       printf( "%4d%21.2f\n", anio, monto );
25    } /* fin de for */
26
27    return 0; /* indica terminación exitosa del programa */
28
29 }  /* fin de la función main */
```

**Figura 4.6** Cálculo del interés compuesto mediante **for**. (Parte 1 de 2.)

```
Anio   Monto del deposito
 1              1050.00
 2              1102.50
 3              1157.63
 4              1215.51
 5              1276.28
 6              1340.10
 7              1407.10
 8              1477.46
 9              1551.33
10              1628.89
```

**Figura 4.6** Cálculo del interés compuesto mediante **`for`**. (Parte 2 de 2.)

**`math.h`**. La función **`pow`** requiere dos argumentos **`double`**. Observe que **`anio`** es un entero. El archivo **`math.h`** incluye información que le indica al compilador que convierta el valor de **`anio`** en una representación temporal **`double`**, antes de llamar a la función. Esta información se encuentra en el *prototipo de la función* **`pow`**. En el capítulo 5, explicaremos los prototipos de función; también incluiremos un resumen de la función **`pow`** y de otras funciones matemáticas de la biblioteca.

Observe que declaramos las variables **`monto`**, **`principal`** y **`tasa`** como de tipo **`double`**. Hicimos esto por simplicidad, ya que estamos manejando partes fraccionarias de dinero.

**Tip para prevenir errores 4.4**

*No utilice variables de tipo **`float`** o **`double`** para realizar cálculos monetarios. La imprecisión de los números de punto flotante puede ocasionar errores que provoquen valores monetarios incorrectos. [En los ejercicios, exploramos el uso de enteros para realizar dichos cálculos.]*

Aquí le presentamos una sencilla explicación sobre lo que puede salir mal si utilizamos **`float`** o **`double`** para representar cantidades en dinero.

Dos montos en dinero de tipo **`float`**, almacenados en la máquina podrían ser 14.234 (lo cual, con **`%.2f`** se mostraría como 14.23), y 18.673 (lo cual, con **`%.2f`** se mostraría como 18.67). Cuando se suman estas cantidades, se produce el resultado 32.907, lo cual, con **`%.2f`**, se mostraría como 32.91. Por lo tanto, su listado podría aparecer como

```
  14.23
+ 18.67
  -----
  32.91
```

sin embargo, ¡es claro que la suma de los números individuales debería ser 32.90! Está usted advertido.

En el programa, utilizamos el especificador de conversión **`%21.2f`** para imprimir el valor de la variable monto. El **`21`** que aparece en el especificador de conversión denota el *ancho del campo* en el que el valor se imprimirá. Un ancho de campo de **`21`** especifica que el valor impreso aparecerá en **`21`** posiciones. El **`2`** especifica la precisión (es decir, el número de posiciones decimales). Si el número de caracteres desplegado es menor que el ancho del campo, entonces el valor se *justificará* automáticamente *a la derecha* del campo. Esto es particularmente útil para alinear valores de punto flotante que tengan la misma precisión (por lo que sus puntos decimales estarán alineados verticalmente). Para justificar hacia la izquierda un valor en el campo, coloque un − (signo −) entre el **`%`** y el ancho del campo. Observe que el signo de menos también puede utilizarse para justificar enteros hacia la izquierda (como en **`%-6d`**) y cadenas de caracteres (como en **`%-8s`**). En el capítulo 9 explicaremos con detalle las poderosas capacidades de formato de **`printf`** y **`scanf`**.

## 4.7 Instrucción de selección múltiple, `switch`

En el capítulo 3, explicamos la instrucción de selección simple **`if`** y la instrucción de selección doble **`if…else`**. En ocasiones, un algoritmo contiene series de decisiones en las que se evalúan una variable o expresión de manera separada para cada uno de los valores integrales constantes que puede asumir, y se llevan a cabo dife-

rentes acciones. A esto se le llama selección múltiple. C proporciona la instrucción de selección múltiple switch, para manejar la toma de decisiones.

La instrucción **switch** consiste en una serie de etiquetas **case** y un caso opcional **default**. La figura 4.7 utiliza la instrucción **switch** para contar el número de cada letra (calificación) diferente que obtuvieron los estudiantes en un examen.

```
/* Figura 4.7: fig04_07.c
   Cuenta las calificaciones expresadas en letras */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int calificacion;   /* una calificación */
   int cuentaA = 0;    /* número de As */
   int cuentaB = 0;    /* número de Bs */
   int cuentaC = 0;    /* número de Cs */
   int cuentaD = 0;    /* número de Ds */
   int cuentaF = 0;    /* número de Fs */

   printf(  “Introduzca la letra que corresponde a la calificacion.\n”  );
   printf(  “Introduzca el caracter EOF para finalizar la entrada de datos.\n”  );

   /* repite hasta que el usuario digita la secuencia de teclas de fin
      de archivo */
  while ( ( calificacion = getchar() ) != EOF ) {

      /* determina cuál calificación se introdujo */
      switch ( calificacion ) { /* switch anidado dentro del while */

         case ‘A’: /* la calificación es A */
         case ‘a’: /* o a */
            ++cuentaA; /* incrementa cuentaA */
            break; /* necesario para salir de switch */

         case ‘B’: /* la calificación es B */
         case ‘b’: /* o b */
            ++cuentaB; /* incrementa cuentaB */
            break; /* sale de switch */

         case ‘C’: /* la calificación es C */
         case ‘c’: /* o c */
            ++cuentaC; /* incrementa cuentaC */
            break; /* sale de switch */

         case ‘D’: /* la calificación es D */
         case ‘d’: /* o d */
            ++cuentaD; /* incrementa cuentaD */
            break; /* sale de switch */

         case ‘F’: /* la calificación es F */
         case ‘f’: /* o f */
            ++cuentaF; /* incrementa cuentaF */
            break; /* sale de switch */
```

**Figura 4.7** Ejemplo de **switch**. (Parte 1 de 2.)

```
48
49             case '\n': /* ignora nuevas líneas, */
50             case '\t': /* tabuladores, */
51             case ' ':  /* y espacios en la entrada */
52                break; /* fin de switch */
53
54             default: /* atrapa todos los demás caracteres */
55                printf( "Introdujo una letra incorrecta." );
56                printf( " Introduzca una nueva calificacion.\n" );
57                break; /* opcional; de todas maneras saldrá del switch */
58          } /* fin de switch */
59
60       } /* fin de while */
61
62       /* muestra el resumen de los resultados */
63       printf( "\nLos totales por calificacion son:\n" );
64       printf( "A: %d\n", cuentaA ); /* despliega el número de calificaciones A */
65       printf( "B: %d\n", cuentaB ); /* despliega el número de calificaciones B */
66       printf( "C: %d\n", cuentaC ); /* despliega el número de calificaciones C */
67       printf( "D: %d\n", cuentaD ); /* despliega el número de calificaciones D */
68       printf( "F: %d\n", cuentaF ); /* despliega el número de calificaciones F */
69
70       return 0; /* indica terminación exitosa del programa */
71
72    } /* fin de la función main */
```

```
Introduzca la letra que corresponde a la calificacion.
Introduzca el caracter EOF para finalizar la entrada de datos.
a
b
c
C
A
d
f
C
E
Introdujo una letra incorrecta. Introduzca una nueva calificacion.
D
A
b
Los totales por calificacion son:
A: 3
B: 2
C: 3
D: 2
F: 1
```

**Figura 4.7** Ejemplo de **switch**. (Parte 2 de 2.)

En el programa, el usuario introduce las calificaciones de un grupo, expresadas con letras. En el encabezado **while** (línea 19),

```
while ( ( calificacion = getchar( ) ) != EOF )
```

la asignación que se encuentra entre paréntesis, **`( calificacion = getchar( ) )`**, se ejecuta primero. La función **`getchar`** (de la biblioteca estándar de entrada/salida) lee un carácter introducido desde el teclado y lo almacena en la variable entera **`calificacion`**. En general, los caracteres se almacenan en variables de tipo ***`char`***. Sin embargo, una característica importante de C es que los caracteres pueden almacenarse en cualquier tipo de dato entero, ya que por lo general, en la computadora se representan como enteros de un byte. Por lo tanto, podemos tratar un carácter como entero o como carácter, de acuerdo con su uso. Por ejemplo, la instrucción

```
printf( “El carácter (%c) tiene el valor %d.\n, ‘a’, ‘a’ );
```

utiliza el especificador de conversión **`%c`** y **`%d`** para desplegar el carácter **`a`** y su valor entero, respectivamente. El resultado es

```
El carácter (a) tiene el valor 97.
```

El entero 97 es la representación numérica del carácter en la computadora. Muchas computadoras actuales utilizan el *conjunto de caracteres ASCII* (*American Standard Code for Information Interchange*), en el cual, el 97 representa la letra minúscula **`‘a’`**. En el apéndice D aparece una lista de los caracteres ASCII y sus valores decimales. Es posible leer caracteres por medio de la función **`scanf`**, a través del especificador de conversión **`%c`**.

Las instrucciones de asignación como un todo, en realidad tienen un valor. Éste es el valor que se le asigna a la variable que se encuentra del lado izquierdo del **`=`**. El valor de la expresión de asignación **`calificacion=getchar()`** es el carácter que devuelve **`getchar`**, el cual se le asigna a la variable **`calificacion`**.

El hecho de que las instrucciones de asignación tengan valores, puede ser útil para inicializar con el mismo valor a muchas variables. Por ejemplo,

```
a = b = c = 0;
```

primero evalúa la asignación **`c = 0`** (ya que el operador **`=`** asocia de derecha a izquierda). Después, a la variable **`b`** se le asigna el valor de la asignación **`c = 0`** (que es cero). Posteriormente, a la variable a se le asigna el valor de la asignación **`b = ( c = 0 )`** (que también es cero). En el programa, el valor de la asignación **`calificacion=getchar( )`** se compara con el valor de **`EOF`** (un símbolo cuyo acrónimo significa “fin de archivo”). Nosotros utilizamos **`EOF`** (que normalmente tiene el valor **`-1`**) como el valor centinela. El usuario escribe una combinación de teclas que dependen del sistema para indicar “fin de archivo”; es decir, “No tengo más datos a introducir”. **`EOF`** es una constante entera simbólica definida en el encabezado **`<stdio.h>`** (en el capítulo 6, veremos cómo se declaran las constantes simbólicas). Si el valor asignado a **`calificacion`** es igual que **`EOF`**, el programa termina. En este programa, elegimos representar los caracteres como **`ints`**, ya que **`EOF`** tiene un valor entero (de nuevo, −1).



### Tip de portabilidad 4.1

*La combinación de teclas necesaria para introducir un **`EOF`** (fin de archivo), depende del sistema.*



### Tip de portabilidad 4.2

*Evaluar la constante simbólica **`EOF`** en lugar de −1, hace más portables a los programas. El C estándar establece que **`EOF`** es un valor integral negativo (pero no necesariamente −1). Por lo tanto, **`EOF`** podría tener valores diferentes en distintos sistemas.*

En sistemas UNIX, y en muchos otros, el indicador de **`EOF`** se introduce escribiendo la secuencia

*<Entrar> <Control+d>*

Esta notación significa que oprima la tecla Entrar y después, de manera simultánea, que oprima tanto la tecla **`Control`** como la tecla **`d`**. En otros sistemas, como Windows de Microsoft, el indicador de **`EOF`** puede introducirse escribiendo:

*<Control+z>*

El usuario introduce las calificaciones por medio del teclado. Cuando oprime la tecla Entrar, la función **`getchar`** lee los caracteres, uno a uno. Si el carácter introducido no es **`EOF`**, se introduce la instrucción **`switch`** (línea 22). La palabra reservada **`switch`** es seguida por el nombre de la variable **`calificacion`**, la cual se encuentra entre paréntesis. A ésta se le llama *expresión de control*. El valor de esta expresión se compara con cada

una de las *etiquetas* ***case***. Suponga que el usuario introdujo la letra **C** como calificación. De manera automática, **C** se compara con cada **case** del **switch**. Si se da una coincidencia **(case 'C' :)**, las instrucciones para ese **case** se ejecutan. En el caso de la letra **C**, **cContador** se incrementa en **1** (línea 36), y se sale inmediatamente de la instrucción **switch** por medio de la instrucción **break**.

La instrucción **break** ocasiona que el control del programa proceda con la primera instrucción después de **switch**. La instrucción **break** se utiliza debido a que, de lo contrario, los **case**s de la instrucción **switch** se ejecutarían juntos. Si no se utiliza **break** en algún lugar de la instrucción **switch**, entonces cada vez que ocurra una coincidencia, las instrucciones de los **case**s restantes se ejecutarán. (Esta característica rara vez es útil, sin embargo, ¡es perfecta para programar la canción iterativa *The Twelve Days of Christmas*!). Si no ocurre coincidencia alguna, el caso **default** se ejecuta, y se despliega un mensaje de error.

Cada **case** puede tener una o más acciones. La instrucción **switch** es diferente de todas las demás instrucciones de control, en que **switch** no necesita llaves alrededor de múltiples acciones **case**. la figura 4.8 muestra el diagrama de flujo de la instrucción general de selección múltiple **switch** (con un **break** en cada **case**).

El diagrama de flujo muestra que cada instrucción **break**, al final de cada **case**, ocasiona que el control salga inmediatamente de la instrucción **switch**. De nuevo, observe que (además de los pequeños círculos y flechas) el diagrama de flujo contiene sólo símbolos rectángulo y rombo. Imagine nuevamente que el programador tiene acceso a un gran montón de instrucciones **switch** vacías (representadas como segmentos del diagrama de flujo), tantas como necesite apilar y anidar con otras instrucciones de control, para formar una implementación estructurada del flujo de control de un algoritmo. Y, de nuevo, los rectángulos y los rombos se llenan con acciones y decisiones apropiadas para el algoritmo.

### Error común de programación 4.5

*Olvidar una instrucción* ***break*** *cuando es necesaria en una instrucción* ***switch****, es un error lógico.*

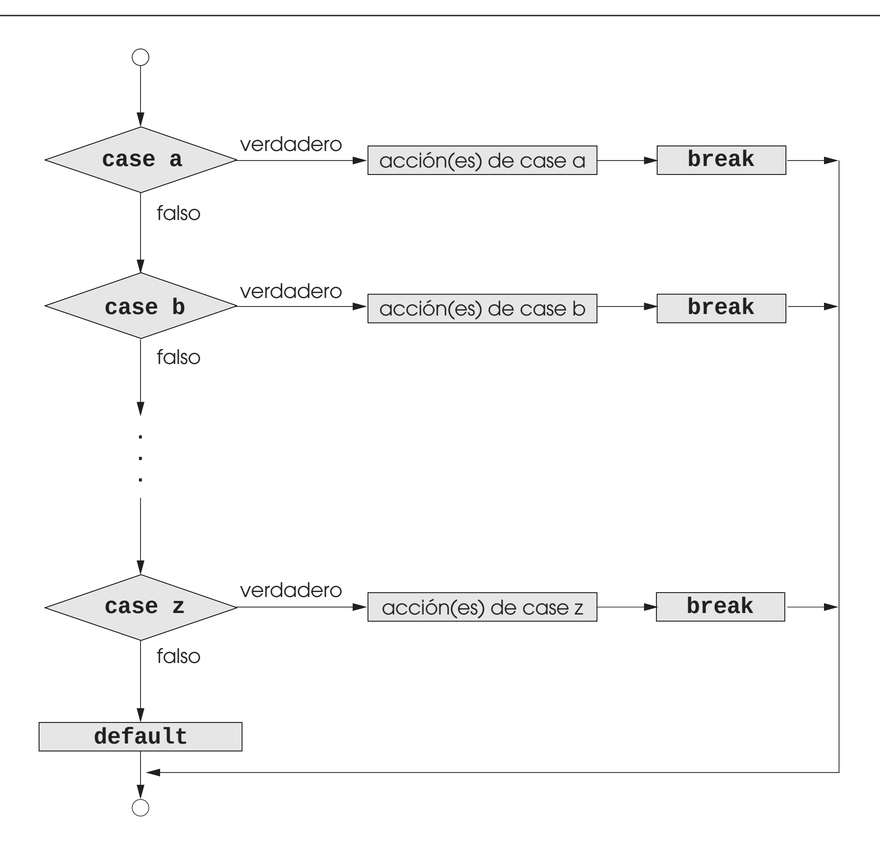



**Figura 4.8** Instrucción de selección múltiple **switch** con **break**s.



### Buena práctica de programación 4.7

*Proporcione un caso* **`default`** *en las instrucciones* **`switch`**. *Los casos no evaluados explícitamente en una instrucción* **`switch`**, *se ignoran. El caso* **`default`** *ayuda a evitar esto, al hacer que el programador se enfoque en la necesidad de procesar condiciones excepcionales. Existen situaciones en las que no se necesita un* **`default`**.



### Buena práctica de programación 4.8

*Aunque las cláusulas* **`case`** *y* **`default`** *de una instrucción* **`switch`** *pueden ocurrir en cualquier orden, colocar la cláusula* **`default`** *al último, se considera una buena práctica de programación.*



### Buena práctica de programación 4.9

*En una instrucción* **`switch`**, *cuando la cláusula* **`default`** *se lista al final, no se necesita una instrucción* **`break`**. *Sin embargo, algunos programadores la incluyen por cuestiones de claridad y simetría con otros* **`case`***s.*

En la instrucción **`switch`** de la figura 4.7, las líneas:

```
case ‘\n’: /* ignora nuevas líneas, */
case ‘\t’: /* tabuladores, */
case ‘\n’: /* y espacios en la entrada */
    break; /* salida de switch */
```

ocasionan que el programa salte las nuevas líneas y los caracteres blancos. Leer los caracteres uno a la vez, puede ocasionar algunos problemas. Para hacer que el programa lea los caracteres, se deben enviar a la computadora oprimiendo la tecla *Entrar*. Esto ocasiona que el carácter nueva línea se coloque en la entrada después del carácter que deseamos procesar. Con frecuencia, esta nueva línea debe procesarse especialmente para hacer que el programa funcione correctamente. Al incluir los casos anteriores en nuestra instrucción **`switch`**, evitamos que el mensaje de error del caso **`default`** se imprima cada vez que una nueva línea o un espacio se encuentren en la entrada.



### Error común de programación 4.6

*No procesar caracteres de nueva línea en la entrada, cuando se leen caracteres uno a uno, puede ocasionar errores lógicos.*



### Tip para prevenir errores 4.5

*Cuando procese caracteres uno por uno, recuerde que debe proporcionar capacidades para procesar nuevas líneas en la entrada.*

Observe que cuando muchas etiquetas case se listan juntas (como en el **`case ‘D’: case ‘d’`**: de la figura 4.7) simplemente significa que se va a llevar a cabo el mismo conjunto de acciones para cualquiera de estos casos.

Cuando utilice la instrucción **`switch`**, recuerde que ésta sólo puede usarse para evaluar una *expresión integral constante*, es decir, cualquier combinación de caracteres y enteros constantes que dan como resultado un valor entero constante. Un carácter constante se representa como el carácter específico entre comillas sencillas, como **`‘A’`**. Los caracteres deben estar encerrados entre comillas sencillas para que sean reconocidos como caracteres constantes. Los enteros constantes son simplemente valores enteros. En nuestro ejemplo, utilizamos caracteres constantes. Recuerde que los caracteres son, en realidad, pequeños valores enteros.

Los lenguajes portables como C deben tener tamaños flexibles para los tipos de datos. Diferentes aplicaciones pueden necesitar enteros de diferentes tamaños. C proporciona diversos tipos de datos para representar enteros. El rango de los valores enteros para cada tipo de dato depende del hardware particular de cada computadora. Además de los tipos **`int`** y **`char`**, C proporciona los tipos **`short`** (una abreviatura de **`short int`**) y **`long`** (una abreviatura de **`long int`**). C especifica que el rango mínimo de valores para enteros **`short`** es $\pm 32767$. Para la gran mayoría de los cálculos con enteros, los enteros **`long`** son suficientes. El estándar especifica que el rango mínimo de valores para enteros **`long`** es $\pm 2147483647$. El estándar establece que el rango de valores para un **`int`** es al menos el mismo que el de los enteros **`short`** y no mayor que el de los enteros **`long`**. El tipo de dato **`char`** puede utilizarse para representar enteros en el rango $\pm 127$ o cualquier carácter del conjunto de caracteres de la computadora.

## 4.8 Instrucción de repetición do…while

La instrucción de repetición **do…while** es parecida a la instrucción **while**. En esta última, la condición de continuación de ciclo se evalúa al principio del ciclo, antes de que el cuerpo de éste se ejecute. La instrucción **do…while** evalúa la condición de continuación de ciclo *después* de que el cuerpo de éste se ejecuta. Por lo tanto, el cuerpo del ciclo se ejecutará al menos una vez. Cuando una **do…while** termina, la ejecución continúa con la instrucción posterior a la cláusula **while**. Observe que en la instrucción **do…while** no es necesario utilizar llaves, si existe sólo una instrucción en el cuerpo. Sin embargo, las llaves por lo general se incluyen para evitar confusiones entre las instrucciones **while** y las **do…while**. Por ejemplo,

**while**( *condición* )

normalmente es considerada como el encabezado de una instrucción **while**. Una **do…while** sin llaves alrededor del cuerpo conformado por una sola instrucción aparece como:

```
do
    instrucción
while( condición );
```

lo cual puede ser confuso. El lector puede malinterpretar la última línea, **while(** *condición* **);** como una instrucción **while** que contiene una instrucción vacía. Por lo tanto, el **do…while** con una sola instrucción se escribe de la siguiente manera para evitar confusiones:

```
do {
    instrucción
}  while( condición );
```



### Buena práctica de programación 4.10

*Algunos programadores siempre incluyen llaves en una instrucción* **do…while**, *incluso si éstas no son necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones* **do…while** *que contienen una instrucción, y las instrucciones* **while**.



### Error común de programación 4.7

*Cuando la condición de continuación de ciclo de una instrucción* **while**, **for** *o* **do…while** *nunca se vuelve falsa, se provoca un ciclo infinito. Para prevenir esto, asegúrese de que no hay un punto y coma inmediatamente después del encabezado de una instrucción* **while** *o de una* **for**. *En un ciclo controlado por contador, asegúrese de que la variable de control se incrementa (o decrementa) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese de que el valor centinela se introduce en algún momento.*

La figura 4.9 utiliza una instrucción **do…while** para desplegar los números del 1 al 10. Observe que la variable de control, **contador**, se preincrementa en la evaluación de continuación de ciclo. También observe el uso de llaves para encerrar el cuerpo de una sola instrucción correspondiente a la instrucción **do…while**.

```
/* Figura 4.9: fig04_09.c
   Uso de la instrucción de repetición do/while */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int contador = 1;              /* inicializa el contador */

   do {
      printf( "%d  ", contador );  /* despliega el contador */
   } while ( ++contador <= 10 );   /* fin del do...while */

```

**Figura 4.9** Ejemplo de la instrucción **do…while**. (Parte 1 de 2.)

```
    return 0; /* indica la terminación exitosa del programa */

} /* fin de la función main */
```

```
1  2  3  4  5  6  7  8  9  10
```

**Figura 4.9** Ejemplo de la instrucción **do...while**. (Parte 2 de 2.)

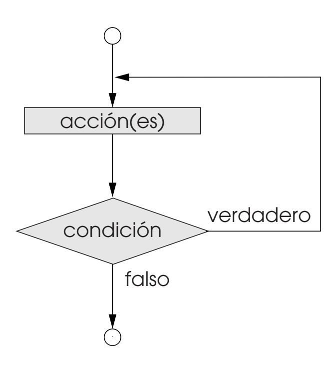



**Figura 4.10** Diagrama de flujo de la instrucción de repetición **do...while**.

El diagrama de flujo de la instrucción **do...while** aparece en la figura 4.10. Este diagrama de flujo muestra que la condición de continuación de ciclo no se ejecuta sino hasta después de que la acción se ejecuta al menos una vez. De nuevo, observe que (además de pequeños círculos y flechas) el diagrama de flujo contiene sólo símbolos rectángulos y rombos. De nuevo, imagine que el programador tiene acceso a un gran montón de instrucciones **do...while** vacías (representadas como segmentos del diagrama de flujo), tantas como necesite apilar y anidar con otras instrucciones de control, para formar una implementación estructurada del flujo de control de un algoritmo. Y, de nuevo, los rectángulos y los rombos se llenan con acciones y decisiones apropiadas para el algoritmo.

## 4.9 Instrucciones break y continue

Las instrucciones **break** y **continue** se utilizan para alterar el flujo de control. La instrucción **break**, cuando se ejecuta en una instrucción **while**, **for**, **do...while** o **switch**, ocasiona la salida inmediata de esa instrucción. La ejecución del programa continúa con la siguiente instrucción. Los usos comunes de la instrucción **break** son: para salir de manera temprana de un ciclo, o para saltar el resto de una instrucción **switch** (como en la figura 4.7). La figura 4.11 muestra la instrucción **break** en una instrucción de repetición **for**. Cuando la instrucción **if** detecta que **x** se ha vuelto **5**, se ejecuta **break**. Esto termina la instrucción **for**, y el programa continúa con la **printf** posterior al **for**. El ciclo se ejecuta completamente, sólo cuatro veces.

```
/* Figura 4.11: fig04_11.c
   Uso de la instrucción break dentro de la instrucción for */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int x; /* contador */
```

**Figura 4.11** Uso de la instrucción **break** en una instrucción **for**. (Parte 1 de 2.)

```

   /* repite 10 veces el ciclo */
   for ( x = 1; x <= 10; x++ ) {

      /* si x es 5, termina el ciclo */
      if ( x == 5 ) {
         break; /* rompe el ciclo sólo si x es 5 */
      } /* fin de if */

      printf( “%d “, x ); /* despliega el valor de x */
   } /* fin de for */

   printf( “\nRompe el ciclo en x == %d\n”, x );

   return 0; /* indica la terminación exitosa del programa */

} /* fin de la función main */
```

```
1 2 3 4
Rompe el ciclo en x == 5
```

**Figura 4.11** Uso de la instrucción **break** en una instrucción **for**. (Parte 2 de 2.)

La instrucción **continue**, cuando se ejecuta en una instrucción **while**, **for** o **do...while**, evita las instrucciones restantes del cuerpo de esa instrucción de control y ejecuta la siguiente iteración del ciclo. En instrucciones **while** y **do...while**, la evaluación de continuación de ciclo se evalúa inmediatamente después de que se ejecuta la instrucción **continue**. En la instrucción **for**, la expresión de incremento se ejecuta, y posteriormente se evalúa la condición de continuación de ciclo. Anteriormente dijimos que la instrucción **while** podía utilizarse en la mayoría de los casos para representar la instrucción **for**. La única excepción ocurre cuando la expresión de incremento de la instrucción **while** se encuentra después de la instrucción **continue**. En este caso, el incremento no se ejecuta antes de que se evalúe la condición de continuación de ciclo, y el **while** no se ejecuta de la misma manera que **for**. La figura 4.12 utiliza la instrucción **continue** en una instrucción **for** para saltar la instrucción **printf**, y continuar con la siguiente iteración del ciclo.

```
/* Figura 4.12: fig04_12.c
   Uso de la instrucción continue dentro de una instrucción for */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int x; /* contador */

   /* repite el ciclo 10 veces */
   for ( x = 1; x <= 10; x++ ) {

      /* si x es 5, continúa con la siguiente iteración del ciclo */
      if ( x == 5 ) {
         continue; /* ignora el resto del código en el cuerpo del ciclo */
      } /* fin de if */

      printf( “%d “, x ); /* despliega el valor de x */
```

**Figura 4.12** Uso de la instrucción **continue** en una instrucción **for**. (Parte 1 de 2.)

```
    } /* fin de for */

    printf( "\nUtiliza continue para ignorar la impresion del valor 5\n" );

    return 0; /* indica la terminación exitosa del programa */

} /* fin de la función main */
```

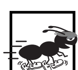
```
1 2 3 4 6 7 8 9 10
Utiliza continue para ignorar la impresion del valor 5
```

**Figura 4.12** Uso de la instrucción **continue** en una instrucción **for**. (Parte 2 de 2.)



**Observación de ingeniería de software 4.2**

*Algunos programadores sienten que las instrucciones* **break** *y* **continue** *violan las normas de la programación estructurada. Debido a que los efectos de estas instrucciones pueden conseguirse por medio de técnicas de programación estructurada que pronto aprenderemos, estos programadores no utilizan* **break** *ni* **continue**.



**Tip de rendimiento 4.1**

*Las instrucciones* **break** *y* **continue***, cuando se utilizan adecuadamente, se ejecutan más rápidamente que las técnicas de programación estructurada correspondientes, que pronto aprenderemos.*



**Observación de ingeniería de software 4.3**

*Existe un conflicto entre lograr una ingeniería de software de calidad y lograr un software con mayor rendimiento. Con frecuencia, uno de estos objetivos se logra a costa del otro.*

## 4.10 Operadores lógicos

Hasta aquí, hemos estudiado sólo *condiciones simples*, como **contador <= 10**, **total > 1000**, y **numero != valorCentinela**. Hemos expresado estas condiciones en términos de operadores de relación, **>**, **<**, **>=** y **<=**, y de operadores de igualdad, **==** y **!=**. Cada decisión evalúa precisamente una condición. Si quisiéramos evaluar diversas condiciones en el proceso de toma de decisiones, tendríamos que ejecutar estas evaluaciones en instrucciones separadas o en instrucciones **if** o **if…else** anidadas.

C proporciona *operadores lógicos* que pueden utilizarse para formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son **&&** (*AND lógico*), **||** (*OR lógico*) y **!** (*NOT lógico*, también conocido como *negación lógica*). Consideraremos ejemplos de cada uno de ellos.

Suponga que deseamos garantizar que dos condiciones sean verdaderas, antes de elegir una cierta ruta de ejecución. En este caso, podemos utilizar el operador lógico **&&** de la siguiente manera:

```
if ( genero == 1 && edad >= 65 )
    ++mujerTerceraEdad;
```

Esta instrucción **if** contiene dos condiciones simples. La condición género **== 1** podría evaluarse, por ejemplo, para determinar si una persona es mujer. La condición **edad >= 65** se evalúa para determinar si una persona es un ciudadano de la tercera edad. Las dos condiciones simples se evalúan primero, debido a que las precedencias de **==** y **>=** son más altas que la precedencia de **&&**. Entonces, la instrucción **if** considera la condición combinada:

```
genero == 1 && edad >= 65
```

Esta condición es verdadera sí y sólo sí ambas condiciones simples son verdaderas. Por último, si esta condición combinada es verdadera, entonces el contador de **mujerTerceraEdad** se incrementa en **1**. Si una o ambas condiciones son falsas, entonces el programa evita el incremento y continúa con la instrucción que se encuentra después de **if**.

| expresión1 | expresión2 | expresión1&&expresión2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | diferente de cero | 0 |
| diferente de cero | 0 | 0 |
| diferente de cero | diferente de cero | 1 |

**Figura 4.13** Tabla de verdad para el operador **&&** (AND lógico).

La figura 4.13 resume el operador **&&**. La tabla muestra las cuatro combinaciones posibles de valores cero (falso) y diferentes de cero (verdadero) para *expresión1* y *expresión2*. Con frecuencia, a dichas tablas se les conoce como *tablas de verdad*. C arroja 0 o 1 para todas las expresiones que incluyen operadores de relación, de igualdad, y/o lógicos. Aunque C establece un 1 a un valor verdadero, acepta *cualquier* valor diferente de cero como verdadero.

Ahora consideremos el operador **||** (OR lógico). Suponga que deseamos garantizar que en algún punto del programa una *o* las dos condiciones sean verdaderas, antes de elegir una cierta ruta de ejecución. En este caso, utilizamos el operador **||** como en el siguiente segmento de programa:

```
if ( promedioSemestral >= 90 || examenFinal >= 90 )
   printf( “La calificación del estudiante es A\n” );
```

Esta instrucción también contiene dos condiciones simples. La condición **promedioSemestral >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un buen desempeño a lo largo del semestre. La condición **examenFinal >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un resultado sobresaliente en el examen final. Entonces, la instrucción **if** considera la condición combinada

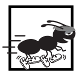
```
promedioSemestral >= 90 || examenFinal >= 90
```

y premia al estudiante con una “A”, si alguna o ambas condiciones simples son verdaderas. Observe que el mensaje “**La calificación del estudiante es A**” no se despliega, únicamente cuando ambas condiciones simples son falsas (cero). La figura 4.14 es una tabla de verdad para el operador lógico OR (**||**).

El operador **&&** tiene una precedencia más alta que **||**. Ambos operadores asocian de izquierda a derecha. Una expresión que contiene los operadores **&&** o **||** se evalúa, sólo hasta que se conozca su verdad o su falsedad. Por lo tanto, la evaluación de la condición

```
genero == 1 && edad >= 65
```

se detendrá, si **genero** es diferente de **1** (es decir, si la expresión completa es falsa), y continuará si **genero** es igual que **1** (es decir, la expresión completa podrá seguir siendo verdadera si **edad >= 65**).



**Tip de rendimiento 4.2**

*En expresiones que utilizan el operador **&&**, haga que la condición más propensa a ser falsa se encuentre hasta la izquierda. En expresiones que utilizan el operador **||**, haga que la condición más propensa a ser verdadera se encuentre hasta la izquierda. Esto puede reducir el tiempo de ejecución de un programa.*

| expresión1 | expresión2 | expresión1 \|\| expresión2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | diferente de cero | 1 |
| diferente de cero | 0 | 1 |
| diferente de cero | diferente de cero | 1 |

**Figura 4.14** Tabla de verdad para el operador lógico OR (**||**).

| expresión | !expresión |
|---|---|
| 0 | 1 |
| diferente de cero | 0 |

**Figura 4.15** Tabla de verdad para el operador **!** (negación lógica).

C proporciona el operador **!** (negación lógica) para permitir al programador "invertir" el significado de una condición. A diferencia de los operadores **&&** y **||**, los cuales combinan dos condiciones (y que, por lo tanto, son operadores binarios), el operador de negación lógica tiene sólo una condición como operando (y por lo tanto, es un operador unario). Cuando estamos interesados en elegir una ruta de ejecución, el operador de negación lógica se coloca antes de una condición, si la condición original (sin el operador de negación lógica) es falsa, como en el siguiente segmento de programa:

```
if ( !( calificacion == valorCentinela ) )
    printf( "La siguiente calificacion es %f\n", calificacion);
```

Los paréntesis alrededor de la condición **calificacion == valorCentinela** son necesarios, ya que el operador de negación lógica tiene una precedencia más alta que el operador de igualdad. La figura 4.15 presenta una tabla de verdad para el operador de negación lógica.

En la mayoría de los casos, el programador puede evitar el uso de la negación lógica, expresando la condición de manera diferente mediante un operador de relación apropiado. Por ejemplo, la instrucción anterior también puede escribirse como:

```
if ( calificacion != valorCentinela )
   printf( "La siguiente calificacion es %f\n", calificacion );
```

La figura 4.16 muestra la precedencia y la asociatividad de los diferentes operadores presentados hasta este punto. Los operadores aparecen de arriba hacia abajo, en orden decreciente de precedencia.

## 4.11 La confusión entre los operadores de igualdad (==) y los de asignación (=)

Existe un tipo de error que los programadores en C, sin importar cuánta experiencia tengan, tienden a cometer con tanta frecuencia, que sentimos que vale la pena una sección especial. Ese error consiste en intercambiar accidentalmente los operadores **==** (de igualdad) y **=** (de asignación). Lo que hace que estos intercambios sean tan dañinos es el hecho de que de manera ordinaria no ocasionan errores de sintaxis. En su lugar, las instruc-

| Operadores | | | | | | Asociatividad | Tipo |
|---|---|---|---|---|---|---|---|
| `++` | `--` | `+` | `-` | `!` | (tipo) | derecha a izquierda | unario |
| `*` | `/` | `%` | | | | izquierda a derecha | multiplicativo |
| `+` | `-` | | | | | izquierda a derecha | aditivo |
| `<` | `<=` | `>` | `>=` | | | izquierda a derecha | de relación |
| `==` | `!=` | | | | | izquierda a derecha | de igualdad |
| `&&` | | | | | | izquierda a derecha | AND lógico |
| `\|\|` | | | | | | izquierda a derecha | OR lógico |
| `?:` | | | | | | derecha a izquierda | condicional |
| `=` | `+=` | `-=` | `*=` | `/=` | `%=` | derecha a izquierda | de asignación |
| `,` | | | | | | izquierda a derecha | coma |

**Figura 4.16** Precedencia y asociatividad de operadores.

ciones con estos errores tienden a compilarse correctamente, lo que permite a los programas ejecutarse en su totalidad, pero es probable que generen resultados incorrectos ocasionados por errores lógicos en tiempo de ejecución.

Existen dos aspectos de C que ocasionan estos problemas. Uno es que cualquier expresión de C que produce un valor, puede utilizarse en la parte de decisión de cualquier instrucción de control. Si el valor es 0, se trata como falso, y si el valor es diferente de cero, se trata como verdadero. El segundo es que las asignaciones en C producen un valor, a saber, el valor que se le asigna a la variable que se encuentra del lado izquierdo del operador de asignación. Por ejemplo, suponga que intentamos escribir

```
if( codigoPago == 4 )
   printf( "¡Usted ganó un bono!" );
```

pero accidentalmente escribimos

```
if( codigoPago == 4 )
   printf( "¡Usted ganó un bono!" );
```

La primera instrucción **if** otorga adecuadamente un bono a la persona cuyo codigoPago es igual que 4. La segunda instrucción **if**, la que contiene el error, evalúa la expresión de asignación en la condición de **if**. Esta expresión es una simple asignación cuyo valor es la constante 4. Debido a que todo valor diferente de cero se interpreta como "verdadero", la condición de esta instrucción **if** siempre es verdadera, y la persona siempre recibe un bono, ¡independientemente del código del pago!



**Error común de programación 4.8**

*Utilizar el operador* **==** *para una asignación, o utilizar el operador* **=** *para una igualdad, es un error lógico.*

Los programadores normalmente escriben condiciones como **x == 7** con el nombre de la variable a la izquierda y la constante a la derecha. Si invertimos esto para que la constante quede a la izquierda y el nombre de la variable a la derecha, como en **7 == x**, el programador que accidentalmente reemplaza el operador **==** por **=**, será protegido por el compilador. El compilador tratará esto como un error de sintaxis, ya que el nombre de una variable sólo puede colocarse en el lado izquierdo de una instrucción de asignación. Al menos, esto evitará la potencial devastación de un error lógico en tiempo de ejecución.

Se dice que los nombres de las variables son *lvalues* (por "valores izquierdos"), ya que pueden utilizarse en el lado izquierdo de un operador de asignación. Se dice que las constantes son *rvalues* (por "valores derechos"), ya que pueden utilizarse sólo en el lado derecho de un operador de asignación. Observe que los *lvalues* también pueden utilizarse como *rvalues*, pero no a la inversa.



**Buena práctica de programación 4.11**

*Cuando una expresión de igualdad tiene una variable y una constante, como en* **x == 1**, *algunos programadores prefieren escribir la expresión con la constante del lado izquierdo y el nombre de la variable del derecho, como protección contra el error lógico que ocurre cuando el programador accidentalmente reemplaza el operador* **==** *con* **=**.

El otro lado de la moneda puede ser igualmente desagradable. Suponga que el programador quiere asignar un valor a la variable con una instrucción sencilla como

```
x = 1;
```

pero en lugar de esto escribe

```
x == 1;
```

Éste, tampoco es un error de sintaxis. El compilador simplemente evalúa la expresión condicional. Si **x** es igual que **1**, la condición es verdadera y la expresión devuelve el valor 1. Si **x** es diferente de **1**, la condición es falsa y la expresión devuelve el valor 0. Independientemente del valor que se devuelva, no hay operador de asignación, por lo que el valor simplemente se pierde, y el valor de **x** permanece inalterado, lo que probablemente ocasione un error lógico en tiempo de ejecución. Por desgracia, ¡no tenemos a la mano un truco que le ayude a evitar este problema!



**Tip para prevenir errores 4.6**

*Después de que escriba un programa, haga una búsqueda de todos los* **`=`**, *y verifique que los está utilizando de manera adecuada.*

## 4.12 Resumen sobre programación estructurada

Tal como los arquitectos diseñan edificios empleando la sabiduría colectiva de su profesión, así deberían los programadores diseñar sus programas. Nuestro campo es más joven que la arquitectura y nuestra sabiduría colectiva es considerablemente menor. Aprendimos grandes cosas en apenas cinco décadas. Tal vez lo más importante sea que aprendimos que la programación estructurada produce programas más fáciles de entender y, por lo tanto, más fáciles de probar, depurar, modificar, e incluso comprobar en un sentido matemático.

Los capítulos 3 y 4 se concentraron en las instrucciones de control de C. Presentamos cada instrucción con diagramas de flujo y las explicamos de manera individual por medio de ejemplos. Ahora, resumimos los resultados de los capítulos 3 y 4, y presentamos un sencillo conjunto de reglas sobre la formación y propiedades de programas estructurados.

La figura 4.17 resume las instrucciones de control que explicamos en los capítulos 3 y 4. En la figura utilizamos pequeños círculos para indicar el punto de entrada simple y el punto de salida simple de cada instrucción. Conectar símbolos individuales de diagrama de flujo de una manera arbitraria puede dar como resultado programas no estructurados. Por lo tanto, la profesión de computación eligió combinar símbolos de diagrama de flujo para formar un conjunto limitado de instrucciones de control, y construir sólo programas estructurados mediante la combinación adecuada de las instrucciones de control; dicha combinación sólo puede hacerse de dos formas. Por simplicidad, sólo se utilizan instrucciones de control de entrada simple/salida simple; existe sólo una forma de introducir cada instrucción de control y sólo una forma de salir de ellas. Conectar instrucciones de control en secuencia para formar programas estructurados es sencillo; el punto de salida de una instrucción de control se conecta directamente con el punto de entrada de la siguiente instrucción de control, es decir, en un programa, las instrucciones de control simplemente se colocan una después de otra (a esto le llamamos “apilar instrucciones de control”). Las reglas para formar programas estructurados también permiten que las instrucciones de control estén anidadas.

La figura 4.18 muestra las reglas para formar programas estructurados. Las reglas suponen que el símbolo rectángulo de un diagrama de flujo puede utilizarse para indicar cualquier acción, incluso las de entrada/salida.

Aplicar las reglas de la figura 4.18 siempre da como resultado un diagrama de flujo estructurado con la apariencia de una cuidadosa construcción con bloques. Aplicar repetidamente la regla 2 al diagrama de flujo más sencillo (figura 4.19) resulta en un diagrama de flujo con muchos rectángulos en secuencia (figura 4.20). Observe que la regla 2 genera una pila de instrucciones de control; por lo que a la regla 2 le llamamos *regla de apilado*.

A la regla 3 se le llama regla de anidamiento. Aplicar repetidamente la regla 3 al diagrama de flujo más sencillo da como resultado un diagrama de flujo con instrucciones de control anidadas pulcramente. Por ejemplo, en la figura 4.21, el rectángulo del diagrama de flujo más sencillo primero es reemplazado por una instrucción de selección doble (**`if…else`**). Después, la regla 3 se aplica nuevamente a los dos rectángulos de la instrucción de selección doble, con lo que se reemplaza a estos rectángulos con instrucciones de selección doble. Los símbolos punteados alrededor de cada una de estas instrucciones de selección doble representan el rectángulo que se sustituyó en el diagrama de flujo original.

La regla 4 genera estructuras anidadas más grandes, más relacionadas y más profundas. Los diagramas de flujo que surgen de la aplicación de las reglas que aparecen en la figura 4.18, constituyen el conjunto de todos los diagramas de flujo estructurados, y por lo tanto, de todos los programas estructurados posibles.

El hecho de que estos bloques de construcción nunca se traslapen, se debe a la eliminación de la instrucción **`goto`**. La belleza del método estructurado es que sólo utilizamos un pequeño número de piezas sencillas de entrada simple/salida simple, y que las ensamblamos de dos sencillas formas. La figura 4.22 muestra las clases de bloques de construcción apilados que surgen de aplicar la regla 2, y las clases de bloques de construcción anidados que surgen de aplicar la regla 3. La figura también muestra la clase de bloques de construcción traslapados que no pueden aparecer en diagramas de flujo estructurados (debido a la eliminación de la instrucción **`goto`**).

Si se siguen las reglas de la figura 4.18, no es posible crear un diagrama de flujo no estructurado (como el de la figura 4.23). Si no está seguro de que un diagrama de flujo en particular sea estructurado, aplique de ma-

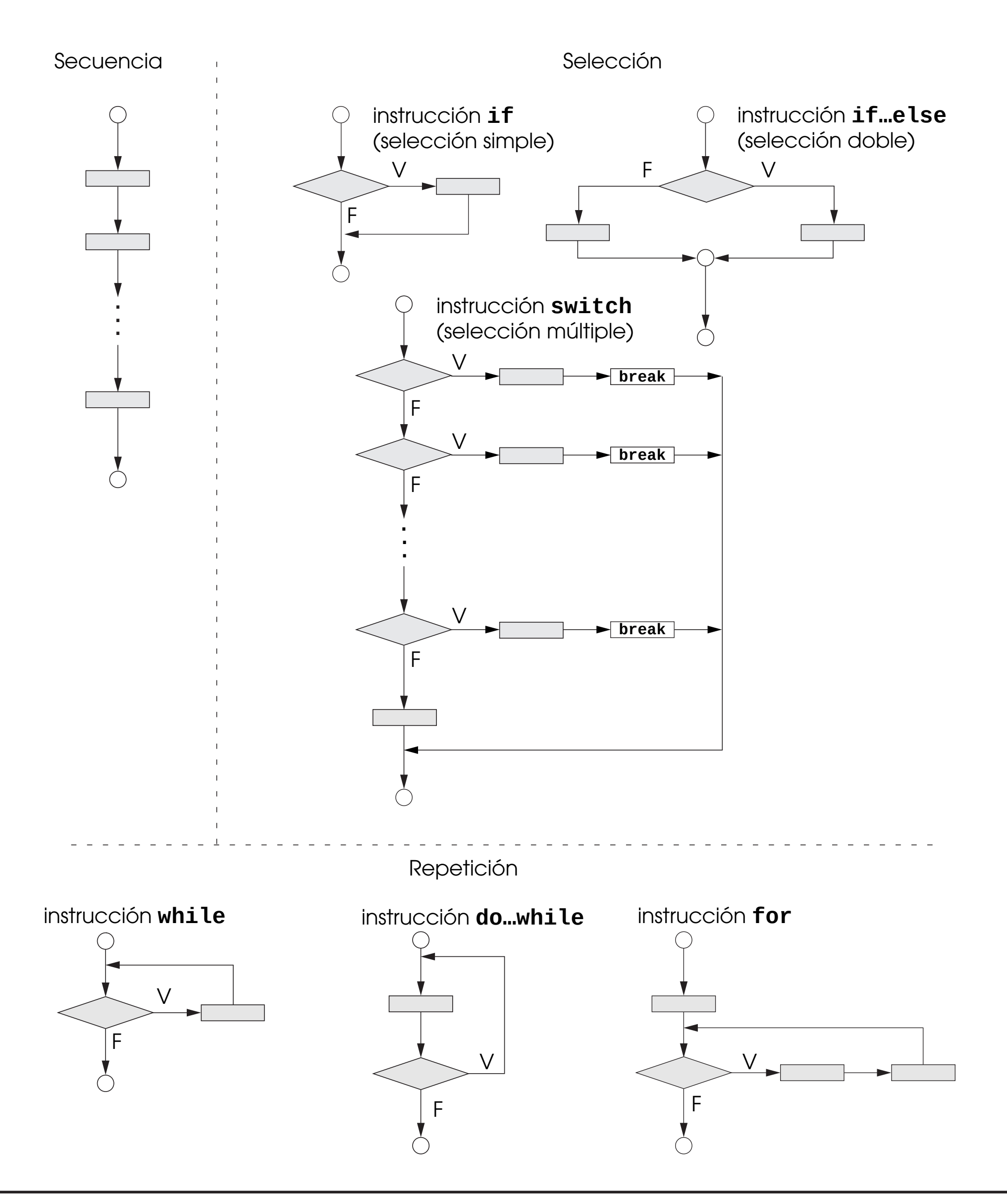



**Figura 4.17** Instrucciones de repetición, selección y secuencia de entrada simple/salida simple de C.

**Reglas para formar programas estructurados**

1) Comience con el “diagrama de flujo más sencillo” (figura 4.19).
2) Cualquier rectángulo (acción) puede ser reemplazada por dos rectángulos (acciones) en secuencia.
3) Cualquier rectángulo (acción) puede ser reemplazado por cualquier instrucción de control (secuencia, `if`, `if…else`, `switch`, `while`, `do…while` o `for`).
4) Las reglas 2 y 3 pueden aplicarse con tanta frecuencia como desee, y en cualquier orden.

**Figura 4.18** Reglas para formar programas estructurados.

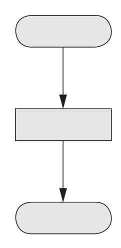


**Figura 4.19** Diagrama de flujo más sencillo.

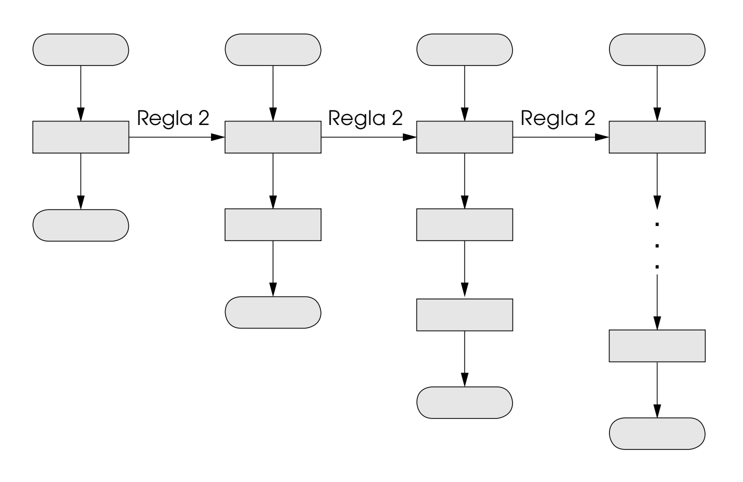



**Figura 4.20** Aplicación repetida de la regla 2 de la figura 4.18 al diagrama de flujo más sencillo.

nera inversa las reglas de la figura 4.18, para intentar reducir el diagrama de flujo a la forma más sencilla. Si el diagrama es reducible al diagrama de flujo más sencillo, entonces es estructurado; de otra manera no lo es.

La programación estructurada promueve la simplicidad. Bohm y Jacopini mostraron que sólo tres formas de control son necesarias:

- Secuencia.
- Selección.
- Repetición.

La secuencia es directa. La selección se implementa en una de las siguientes formas:

- Instrucción **`if`** (selección simple).
- Instrucción **`if…else`** (selección doble).
- Instrucción **`switch`** (selección múltiple).

De hecho, es fácil demostrar que la instrucción simple **`if`** es suficiente para proporcionar cualquier forma de selección; todo lo que puede hacerse con las instrucciones **`if…else`** y **`switch`** puede implementarse con una o más instrucciones **`if`**.

La repetición se implementa en una de las tres siguientes formas:

- Instrucción **`while`**.
- Instrucción **`do…while`**.
- Instrucción **`for`**.

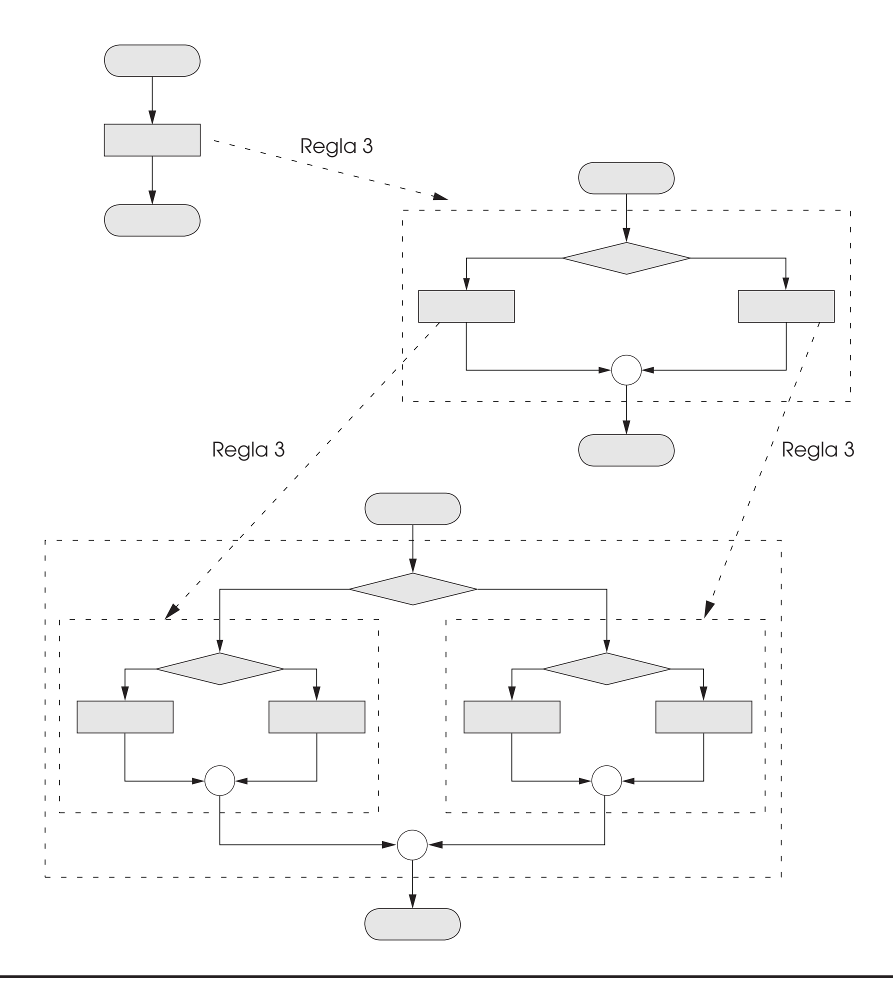



**Figura 4.21** Aplicación de la regla 3 de la figura 4.18 al diagrama de flujo más sencillo.

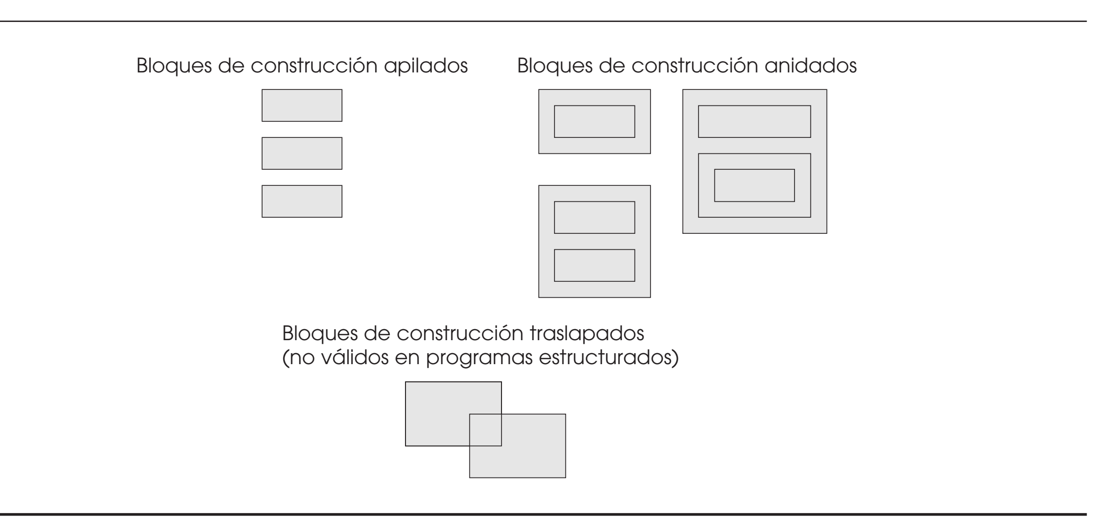



**Figura 4.22** Bloques de construcción apilados, anidados y traslapados.

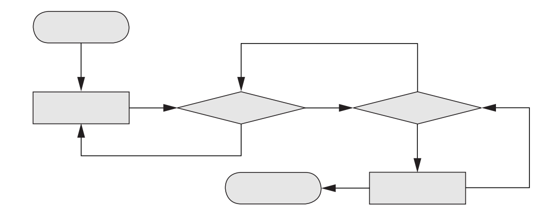


**Figura 4.23** Diagrama de flujo no estructurado.

Es fácil demostrar que la instrucción **while** es suficiente para proporcionar cualquier forma de repetición. Todo lo que puede hacerse con las instrucciones **do…while** y **for** puede hacerse con la instrucción **while**.

La combinación de estos resultados ilustra que cualquier forma de control necesaria en un programa en C, puede expresarse en términos de sólo tres formas de control:

- Secuencia.
- Instrucción **if** (selección).
- Instrucción **while** (repetición).

Y, estas instrucciones de control pueden combinarse en sólo dos formas: apilamiento y anidamiento. De hecho, la programación estructurada promueve la simplicidad.

En los capítulos 3 y 4, explicamos cómo elaborar programas a partir de instrucciones de control que contienen acciones y decisiones. En el capítulo 5, presentamos otra unidad para estructurar programas, llamada *función*. También explicaremos cómo el hecho de utilizar funciones promueve la reutilización de software.

## RESUMEN

- Un ciclo es un conjunto de instrucciones que la computadora ejecuta repetidamente, hasta que una condición de terminación se satisface. Dos formas de repetición son: la controlada por contador y la controlada por centinela.
- Un contador de ciclo se utiliza para contar el número de veces que debe repetirse un grupo de instrucciones. Éste se incrementa (normalmente en 1) cada vez que el grupo de instrucciones se ejecuta.
- Los valores centinela se utilizan generalmente para controlar una repetición en la que no se conoce por adelantado el número preciso de repeticiones, y el ciclo incluye instrucciones para obtener los datos cada vez que el ciclo se ejecuta.
- Un valor centinela se introduce después de que todos los datos regulares se le han proporcionado al programa. Los centinelas deben elegirse cuidadosamente para que no exista posibilidad alguna de confundirlos con datos válidos.
- La instrucción de repetición **for** maneja todos los detalles de la repetición controlada por contador. La forma general de la instrucción **for** es

  **for (** *expresión1* **;** *expresión2* **;** *expresión3* **)**
  
  *instrucción*

  donde *expresión1* inicializa la variable de control del ciclo, *expresión2* es la condición de continuación del ciclo, y *expresión3* incrementa (o decrementa) la variable de control.
- La instrucción de repetición **do…while** es parecida a la instrucción de repetición **while**, pero la primera evalúa la condición de repetición de ciclo al final del ciclo, de tal forma que el ciclo se ejecutará al menos una vez. La forma de la instrucción **do…while** es

  **do**
  
  *instrucción*
  
  **while (** *condición* **);**
- La instrucción **break**, cuando se ejecuta en una de las instrucciones de repetición (**for**, **while** y **do…while**), ocasiona la salida inmediata de la instrucción. La ejecución continúa con la primera instrucción después del ciclo. La instrucción **break** también puede utilizarse para salir de una instrucción **switch**.
- La instrucción **continue**, cuando se ejecuta en una de las instrucciones de repetición (**for**, **while** y **do…while**), salta cualquier instrucción restante del cuerpo de la instrucción de control, y continúa con la siguiente iteración del ciclo.

- La instrucción **switch** maneja una serie de decisiones en las que una variable o expresión en particular se evalúa con cada uno de los valores que puede asumir, y se toman diferentes acciones. Cada case de una instrucción **switch** puede ocasionar que se ejecuten muchas instrucciones. En la mayoría de los programas es necesario incluir un **break** después de las instrucciones de cada **case**, de otro modo, el programa ejecutará las instrucciones de cada **case** hasta que encuentre un **break**, o hasta que alcance el final de la instrucción **switch**. Muchos **case**s pueden ejecutar las mismas instrucciones, listando las etiquetas **case** antes de las instrucciones. La instrucción **switch** sólo puede evaluar expresiones integrales constantes.
- La función **getchar** devuelve un carácter proveniente del teclado (la entrada estándar) como un entero.
- En sistemas UNIX y en muchos otros, el carácter **EOF** se introduce escribiendo la secuencia

  *<Entrar> <Control+d>*

  En sistemas Windows de Microsoft, el carácter **EOF** se introduce escribiendo

  *<Control+z>*
- Los operadores lógicos pueden utilizarse para formar condiciones complejas, mediante la combinación de condiciones. Los operadores lógicos son **&&**, **||** y **!**, que significan AND lógico, OR lógico y NOT lógico (negación), respectivamente.
- Un valor verdadero es cualquier valor diferente de cero.
- Un valor falso es 0 (cero).

## TERMINOLOGÍA

ancho de campo
AND lógico (**&&**)
caso **default** de una instrucción **switch**
ciclo infinito
condición de continuación de ciclo
condición simple
conjunto de caracteres ASCII
contador de ciclo
*<Control+z>*
cuerpo de un ciclo
**char**
decremento de la variable de control
**double**
*<Entrar> <Control+d>*
**EOF**
error por desplazamiento en uno
etiqueta **case**
fin de archivo
función **getchar**
función **pow**
incremento de la variable de control
instrucción de control **break**
instrucción de control **continue**
instrucción de repetición **do…while**
instrucción de repetición **for**
instrucción de repetición **while**
instrucción de selección **switch**
instrucciones de control anidadas
instrucciones de control de entrada simple/salida simple
instrucciones de repetición
justificación hacia la derecha
justificación hacia la izquierda
**long**
*lvalue* (“valores izquierdos”)
negación lógica (**!**)
operador unario
operadores lógicos
OR lógico (**||**)
regla de anidamiento
regla de apilamiento
repetición controlada por contador
repetición definida
repetición indefinida
*rvalue* (“valores derechos”)
selección múltiple
**short**
signo menos para justificación a la izquierda
tabla de verdad
valor final de la variable de control
valor inicial de la variable de control
variable de control
variable de control de ciclo

## ERRORES COMUNES DE PROGRAMACIÓN

**4.1** Debido a que los valores de punto flotante pueden ser aproximados, controlar ciclos contadores con variables de punto flotante puede dar como resultado valores contadores imprecisos y evaluaciones de terminación incorrectas.

**4.2** Utilizar un operador de relación incorrecto o usar un valor final incorrecto en un contador de ciclo, dentro de la condición de una instrucción **while** o **for**, puede ocasionar errores por desplazamiento en uno.

**4.3** Utilizar comas en lugar de puntos y comas en un encabezado **for**, es un error de sintaxis.

**4.4** Colocar un punto y coma inmediatamente a la derecha del paréntesis de un encabezado **for**, convierte el cuerpo de dicha instrucción en una instrucción vacía. Por lo general, éste es un error lógico.

**4.5** Olvidar una instrucción **break** cuando es necesaria en una instrucción **switch**, es un error lógico.

**4.6** No procesar caracteres de nueva línea en la entrada, cuando se leen caracteres uno a uno, puede ocasionar errores lógicos.

**4.7** Cuando la condición de continuación de ciclo de una instrucción **while**, **for** o **do…while** nunca se vuelve falsa, se provoca un ciclo infinito. Para prevenir esto, asegúrese de que no hay un punto y coma inmediatamente después

del encabezado de una instrucción **while** o de una **for**. En un ciclo controlado por contador, asegúrese de que la variable de control se incrementa (o decrementa) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese de que el valor centinela se introduce en algún momento.

**4.8** Utilizar el operador **==** para una asignación, o utilizar el operador **=** para una igualdad, es un error lógico.

## *TIPS PARA PREVENIR ERRORES*

**4.1** Controle los ciclos contadores con valores enteros.

**4.2** Utilizar el valor final en la condición de una instrucción **while** o **for**, y utilizar el operador de relación **<=**, ayudará a evitar errores por desplazamiento en uno. Por ejemplo, para un ciclo utilizado para imprimir los valores del 1 al 10, la condición de continuación de ciclo debe ser **contador <= 10**, en lugar de **contador < 11** o **contador < 10**.

**4.3** Aunque el valor de la variable de control puede modificarse en el cuerpo de un ciclo **for**, esto puede provocar errores sutiles. Es mejor no cambiarlo.

**4.4** No utilice variables de tipo **float** o **double** para realizar cálculos monetarios. La imprecisión de los números de punto flotante puede ocasionar errores que provoquen valores monetarios incorrectos. [En los ejercicios, exploramos el uso de enteros para realizar dichos cálculos.]

**4.5** Cuando procese caracteres uno por uno, recuerde que debe proporcionar capacidades para procesar nuevas líneas en la entrada.

**4.6** Después de que escriba un programa, haga una búsqueda de todos los **=**, y verifique que los está utilizando de manera adecuada.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**4.1** Sangre las instrucciones correspondientes al cuerpo de toda instrucción de control.

**4.2** Coloque una línea en blanco antes y después de cada instrucción de control, para que resalten en el programa.

**4.3** Tener demasiados niveles de anidamiento, puede provocar que un programa sea difícil de entender. Como regla general, intente evitar el uso de más de tres niveles de anidamiento.

**4.4** Combinar espaciado vertical, antes y después de las instrucciones de control, con sangría en los cuerpos de dichas instrucciones, proporciona a los programas una apariencia bidimensional, la cual mejora bastante la legibilidad del programa.

**4.5** Aunque las instrucciones que preceden a **for** y las instrucciones del cuerpo de un **for**, a menudo se pueden fusionar dentro de un encabezado **for**, evite hacerlo, ya que esto ocasiona que el programa sea más difícil de leer.

**4.6** Si es posible, limite el tamaño de los encabezados de las instrucciones de control a una sola línea.

**4.7** Proporcione un caso **default** en las instrucciones **switch**. Los casos no evaluados explícitamente en una instrucción **switch**, se ignoran. El caso **default** ayuda a evitar esto, al hacer que el programador se enfoque en la necesidad de procesar condiciones excepcionales. Existen situaciones en las que no se necesita un **default**.

**4.8** Aunque las cláusulas **case** y **default** de una instrucción **switch** pueden ocurrir en cualquier orden, colocar la cláusula **default** al último, se considera una buena práctica de programación.

**4.9** En una instrucción **switch**, cuando la cláusula **default** se lista al final, no se necesita una instrucción **break**. Sin embargo, algunos programadores la incluyen por cuestiones de claridad y simetría con otros **case**s.

**4.10** Algunos programadores siempre incluyen llaves en una instrucción **do...while**, incluso si éstas no son necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones **do...while** que contienen una instrucción, y las instrucciones **while**.

**4.11** Cuando una expresión de igualdad tiene una variable y una constante, como en **x == 1**, algunos programadores prefieren escribir la expresión con la constante del lado izquierdo y el nombre de la variable del derecho, como protección contra el error lógico que ocurre cuando el programador accidentalmente reemplaza el operador **==** con **=**.

## *TIPS DE RENDIMIENTO*

**4.1** Las instrucciones **break** y **continue**, cuando se utilizan adecuadamente, se ejecutan más rápidamente que las técnicas de programación estructurada correspondientes, que pronto aprenderemos.

**4.2** En expresiones que utilizan el operador **&&**, haga que la condición más propensa a ser falsa se encuentre hasta la izquierda. En expresiones que utilizan el operador **||**, haga que la condición más propensa a ser verdadera se encuentre hasta la izquierda. Esto puede reducir el tiempo de ejecución de un programa.

## TIPS DE PORTABILIDAD

**4.1** La combinación de teclas necesaria para introducir un **EOF** (fin de archivo), depende del sistema.

**4.2** Evaluar la constante simbólica **EOF** en lugar de $-1$, hace más portables a los programas. El C estándar establece que **EOF** es un valor integral negativo (pero no necesariamente $-1$). Por lo tanto, **EOF** podría tener valores diferentes en distintos sistemas.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**4.1** Dentro de las secciones de inicialización e incremento de una instrucción **for**, sólo coloque expresiones relacionadas con las variables de control. La manipulación de otro tipo de variables debe aparecer ya sea antes del ciclo (si se deben ejecutar sólo una vez, como las instrucciones de inicialización), o dentro del cuerpo del ciclo (si se deben ejecutar una vez por repetición, como las instrucciones de incremento y decremento).

**4.2** Algunos programadores sienten que las instrucciones **break** y **continue** violan las normas de la programación estructurada. Debido a que los efectos de estas instrucciones pueden conseguirse por medio de técnicas de programación estructurada que pronto aprenderemos, estos programadores no utilizan **break** ni **continue**.

**4.3** Existe un conflicto entre lograr una ingeniería de software de calidad y lograr un software con mayor rendimiento. Con frecuencia, uno de estos objetivos se logra a costa del otro.

## EJERCICIOS DE AUTOEVALUACIÓN

**4.1** Complete los espacios en blanco.

a) A la repetición controlada por contador también se le conoce como repetición ____________, ya que se sabe por adelantado el número de veces que se ejecutará el ciclo.

b) A la repetición controlada por centinela también se le conoce como repetición ____________, ya que no se sabe por adelantado el número de veces que se ejecutará el ciclo.

c) En la repetición controlada por contador se utiliza un ____________ para contar el número de veces que un grupo de instrucciones debe repetirse.

d) La instrucción ____________, cuando se ejecuta en una instrucción de repetición, ocasiona que se ejecute inmediatamente la siguiente iteración del ciclo.

e) La instrucción ____________, cuando se ejecuta en una instrucción de repetición o en un **switch**, ocasiona la salida inmediata de la instrucción.

f) La ____________ se utiliza para evaluar una variable o expresión en particular para cada uno de los valores integrales constantes que puede asumir.

**4.2** Diga si los siguientes enunciados son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.

a) En la instrucción de selección **switch**, es necesario un caso **default**.

b) La instrucción **break** es necesaria en el caso **default** de una instrucción de selección **switch**.

c) La expresión (**x >** y **&& a < b**) es verdadera si **x** > **y** o si **a** < **b**.

d) Una expresión que contiene el operador **||** es verdadera si uno o ambos de sus operandos son verdaderos.

**4.3** Escriba una instrucción o un conjunto de instrucciones para realizar las siguientes tareas:

a) Sume los enteros impares entre 1 y 99, utilizando una instrucción **for**. Suponga que las variables enteras **suma** y **cuenta** ya fueron declaradas.

b) Imprima el valor **333.546372** en un ancho de campo de **15** caracteres con precisiones de **1**, **2**, **3**, **4** y **5**. Justifique hacia la izquierda la salida. ¿Cuáles son los valores que despliega?

c) Calcule el valor de **2.5** elevado a la tercera potencia, utilizando la función **pow**. Imprima el resultado con una precisión de **2**, en un ancho de campo de **10** posiciones. ¿Cuál es el valor que despliega?

d) Imprima los enteros del 1 al 20, utilizando un ciclo **while** y la variable contador **x**. suponga que la variable **x** ya fue declarada, pero no inicializada. Imprima sólo cinco enteros por línea. [*Pista:* Utilice el cálculo **x % 5**. Cuando el valor de éste sea 0, imprima un carácter de nueva línea, cuando sea diferente imprima un carácter tabulador.]

e) Repita el ejercicio 4.3 (d), utilizando una instrucción **for**.

**4.4** Encuentre el error en cada uno de los siguientes segmentos de código, y explique cómo corregirlos.

a)
```
x = 1;
while ( x <= 10 );
   x++;
}
```
b)
```
for ( y = .1; y != 1.0; y += .1 )
   printf( "%f\n", y );
```
c)
```
switch ( n ) {
   case 1:
      printf( "El número es 1\n" );
   case 2:
      printf( "El número es 2\n" );
      break;
   default:
      printf( "El número no es 1 o 2\n" );
      break;
   }
```
d) El siguiente código debe imprimir los valores del 1 al 10.
```
n= 1;
while ( n < 10 )
   printf( "%d ", n++ );
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**4.1** a) Definida. b) Indefinida. c) Variable de control o contador. d) `continue`. e) `break`. f) Instrucción de selección `switch`.

**4.2** a) Falso. El caso `default` es opcional. Si no es necesaria una acción predeterminada, entonces no se necesita un caso `default`.

b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no es necesaria cuando el caso `default` es el último caso.

c) Falso. Cuando se utiliza el operador **&&**, ambas expresiones de relación deben ser verdaderas para que toda la expresión sea verdadera.

d) Verdadero.

**4.3** a)
```
suma = 0;
for ( cuenta = 1; cuenta <= 99; cuenta +=2 )
   suma += cuenta;
```
b)
```
printf( "%-15.1f\n", 333.546372 ); /* imprime 333.5 */
printf( "%-15.2f\n", 333.546372 ); /* imprime 333.55 */
printf( "%-15.3f\n", 333.546372 ); /* imprime 333.546 */
printf( "%-15.4f\n", 333.546372 ); /* imprime 333.5464 */
printf( "%-15.5f\n", 333.546372 ); /* imprime 333.54637 */
```
c)
```
printf( "%10.2f\n", pow( 2.5, 3 ) ) ; /* imprime 15.63 */
```
d)
```
x = 1;
while( x <= 20 ) {
   printf( "%d", x );
   if ( x % 5 == 0 )
      printf( "\n" );
   else
      printf( "\t" );
   x++;
}
```
o
```
x = 1;
while ( x <= 20 )
   if( x % 5 == 0 )
```

```
      printf( “%d\n”, x++ );
   else
      printf( “%d\t”, x++ );
```

o

```
x = 0;
while ( ++x <= 20 )
   if ( x % 5 == 0 )
      printf( “%d\n”, x );
   else
      printf( “%d\t”, x );
```

e)
```
for ( x = 1; x <= 20; x++ ) {
   printf( “%d”, x );
   if ( x % 5 == 0 )
      printf( “\n” );
   else
      printf( “\t” );
}
```

o

```
for ( x = 1; x <= 20; x++ )
   if ( x % 5 == 0 )
      printf( “%d\n”, x );
   else
      printf( “%d\t”, x );
```

**4.4** a) Error: el punto y coma después del encabezado de `while` ocasiona un ciclo infinito.
Corrección: reemplace el punto y coma por una `{`, o elimine tanto el `;` como la `}`.

b) Error: utilizar un número de punto flotante para controlar una instrucción de repetición `for`.
Corrección: utilice un entero y realice el cálculo adecuado para obtener los valores que desea.

```
for ( y = 1; y != 10; y++ )
printf( “%f\n”, ( float ) y / 10 );
```

c) Error: olvidar la instrucción `break` en las instrucciones para el primer `case`.
Corrección: añada un `break` al final de las instrucciones del primer `case`. Observe que esto no necesariamente es un error, si el programador quiere que la instrucción del `case 2` se ejecute cada vez que el `case 1` se ejecuta.

d) Error: se utilizó un operador de relación incorrecto en la condición de continuación de ciclo `while`.
Corrección: utilice `<=`, en lugar de `<`.

## EJERCICIOS

**4.5** Encuentre el error en cada uno de los siguientes ejercicios (*Nota:* Puede haber más de un error):

a)
```
for ( x = 100, x >= 1, x++ )
   printf( “%d\n”, x );
```

b) El siguiente código debe imprimir si un entero es par o impar:

```
switch ( valor % 2 ) {
   case 0:
      printf( “Entero par\n” );
   case 1:
      printf( “Entero impar\n” );
}
```

c) El siguiente código debe introducir un entero y un carácter e imprimirlos. Suponga que el usuario escribe 100 A.

```
scanf( “%d”, &valorEnt );
valorCarac = getchar( );
printf( “Entero: %d\nCaracter: %c\n”, valorEnt, valorCarac );
```

d) `for ( x = .000001; x <= .0001; x += .000001 )`

`printf("%.7f\n",x);`

e) El siguiente código debe desplegar los enteros impares del 999 al 1:

```
for ( x = 999; x>= 1; x += 2 )
   printf( "%d\n", x );
```

f) El siguiente código debe desplegar los números pares del 2 al 100:

```
contador = 2;

Do {
   if ( contador % 2 == 0 )
      printf( "%d\n", contador );

contador += 2;
} While ( contador < 100 );
```

g) El siguiente código debe sumar los enteros del 100 al 150 (suponga que **total** se inicializó en 0):

```
for ( x = 100; x <= 150; x++ );
   total += x;
```

**4.6** Establezca cuáles valores de la variable de control son desplegados por cada una de las siguientes instrucciones:

a)
```
for ( x = 2; x <= 13; x += 2 )
   printf( "%d\n", x );
```
b)
```
for ( x = 5; x <= 22; x += 7 )
   printf( "%d\n", x );
```
c)
```
for ( x = 3; x <= 15; x += 3 )
   printf( "%d\n", x );
```
d)
```
for ( x = 1; x <= 5; x += 7 )
   printf( "%d\n", x );
```
e)
```
for ( x = 12; x >= 2; x += 3 )
   printf( "%d\n", x );
```

**4.7** Escriba instrucciones **for** que impriman la siguiente secuencia de valores:

a) 1, 2, 3, 4, 5, 6, 7
b) 3, 8, 13, 18, 23
c) 20, 14, 8, 2, −4, −10
d) 19, 27, 35, 43, 51

**4.8** ¿Qué es lo que hace el siguiente programa?

```
/* ej04_08.c
   ¿Qué es lo que imprime el programa? */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int x;
   int y;
   int i;
   int j;

   /* indica al usuario la entrada de datos */
   printf( "Introduzca dos enteros entre 1 y 20: " );
   scanf( "%d%d", &x, &y ); /* lee los valores para x e y */
```

(Parte 1 de 2.)

```
 
   for ( i = 1; i <= y; i++ ) { /* cuenta de 1 a y */
 
      for ( j = 1; j <= x; j++ ) { /* cuenta de 1 a x */
         printf( "@" ); /* imprime @ */
      } /* fin del for interno */
 
      printf( "\n" ); /* inicia una nueva línea */
   } /* fin del for externo */
 
   return 0; /* indica la terminación exitosa del programa */
 
} /* fin de la función main */
```

(Parte 2 de 2.)

**4.9** Escriba un programa que sume una secuencia de enteros. Asuma que el primer entero leído mediante **`scanf`** especifica el número de valores restantes que se introducirán. Su programa debe leer únicamente un valor cada vez que se ejecuta **`scanf`**. Una secuencia de entrada típica podría ser

**`5  100  200  300  400  500`**

donde el **`5`** indica que se sumarán los cinco números subsiguientes.

**4.10** Escriba un programa que calcule e imprima el promedio de varios enteros. Suponga que el último valor que lee la instrucción **`scanf`** es el centinela **`9999`**. Una secuencia de entrada típica podría ser

**`10  8  11  7  9  9999`**

que indica que calculará el promedio de todos los valores que anteceden a **`9999`**.

**4.11** Escriba un programa que encuentre el menor de varios enteros. Suponga que el primer valor a leer especifica el número de valores restantes.

**4.12** Escriba un programa que calcule e imprima la suma de los enteros pares del 2 al 30.

**4.13** Escriba un programa que calcule e imprima el producto de los enteros nones del 1 al 15.

**4.14** A menudo, la función *factorial* se utiliza en problemas de probabilidad. El factorial de un entero positivo *n* (se escribe *n!* y se pronuncia "n factorial") es igual al producto de los enteros positivos de 1 a *n*. Escriba un programa que evalúe los factoriales de los enteros de 1 a 5. Imprima los resultados de manera tabular. ¿Qué dificultad debe usted prever al calcular el factorial de 20?

**4.15** Modifique el programa del interés compuesto de la sección 4.16 para repetir sus pasos para tasas de interés del 5 por ciento, 6 por ciento, 8 por ciento, 9 por ciento, y 10 por ciento. Utilice un **`for`** para crear un ciclo que varíe la tasa de interés.

**4.16** Escriba un programa que imprima los patrones siguientes de manera separada, uno debajo del otro. Utilice ciclos **`for`** para generar los patrones. Todos los asteriscos (**`*`**) deben imprimirse mediante una sola instrucción **`printf`** de la forma **`printf("*");`** (esto provoca que los asteriscos se impriman uno al lado del otro). *Pista:* Los dos últimos patrones requieren que cada línea comience con el número apropiado de espacios en blanco.

```
(A)         (B)         (C)         (D)
*           **********  **********           *
**          *********    *********          **
***         ********      ********         ***
****        *******        *******        ****
*****       ******          ******       *****
******      *****            *****      ******
*******     ****              ****     *******
********    ***                ***    ********
*********   **                  **   *********
**********  *                    *  **********
```

**4.17** Recuperar el dinero se hace más difícil durante los periodos de recesión, de manera que las empresas deben reducir sus límites de crédito para prevenir que sus cuentas por cobrar (el dinero prestado) se hagan muy grandes. En respuesta a la prolongada recesión, una empresa recortó sus límites de crédito a la mitad. De esta manera, si un cliente en particular tenía un límite de crédito de $2000, ahora su límite es de $1000. Si un cliente tenía un límite de crédito de $5000, este cliente tiene ahora un límite de crédito de $2500. Escriba un programa que analice el estado del crédito de tres clientes de esta empresa. Por cada cliente a usted se le brinda:

a) El número de cuenta del cliente.
b) El límite de crédito antes de la recesión.
c) El saldo actual del cliente (es decir, el monto que le debe el cliente a la empresa).

Su programa debe calcular e imprimir el nuevo límite de crédito para cada cliente, y debe determinar (e imprimir) cuáles clientes tienen saldos que exceden los nuevos límites de crédito.

**4.18** Una interesante aplicación de las computadoras es dibujar gráficos de barras (en ocasiones llamadas “histogramas”). Escriba un programa que lea cinco números (cada uno entre 1 y 30). Por cada número leído, su programa debe imprimir una línea que contenga dicho número con asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe imprimir `*******`.

**4.19** Una empresa de ventas por correo vende cinco productos diferentes cuyos precios de lista mostramos en la siguiente tabla:

| Número de producto | Precio de lista |
|---|---|
| 1 | $2.98 |
| 2 | $4.50 |
| 3 | $9.98 |
| 4 | $4.49 |
| 5 | $6.87 |

Escriba un programa que lea una serie de pares de números de la siguiente manera:

a) Número de producto.
b) Cantidad vendida durante el día.

Su programa debe utilizar una instrucción **`switch`** para ayudar a determinar el precio de lista de cada producto. Su programa debe calcular y desplegar el valor total de venta de todos los productos vendidos la semana pasada.

**4.20** Complete las siguientes tablas de verdad, llenando cada espacio en blanco con un 1 o un 0.

| Condición1 | Condición2 | Condición1 && Condición2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | diferente de cero | 0 |
| diferente de cero | 0 | _____ |
| diferente de cero | diferente de cero | _____ |

| Condición1 | Condición2 | Condición1 \|\| Condición2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | diferente de cero | 1 |
| diferente de cero | 0 | _____ |
| diferente de cero | diferente de cero | _____ |

| Condición1 | !Condición1 |
|---|---|
| 0 | 1 |
| diferente de cero | _____ |

**4.21** Rescriba el programa de la figura 4.2 de manera que la inicialización de la variable contador se haga en la declaración, en lugar de hacerlo en la instrucción **`for`**.

**4.22** Modifique el programa de la figura 4.7 de manera que calcule el promedio de calificaciones del grupo.

**4.23** Modifique el programa de la figura 4.6 de manera que sólo utilice enteros para calcular el interés compuesto. [*Pista:* Trate todas las cantidades monetarias como números enteros de centavos. Luego, "rompa" el resultado en su parte entera y de centavos mediante el uso de las operaciones de división y de residuo, respectivamente. Inserte un punto.]

**4.24** Suponga que **`i=1`**, **`j=2`**, **`k=3`** y **`m=2`**. ¿Qué imprimen cada una de las siguientes instrucciones?

a) **`printf( "%d", j==1 );`**
b) **`printf( "%d", j==3 );`**
c) **`printf( "%d", i >= 1 && j < 4 );`**
d) **`printf( "%d", m >= 99 && k < m );`**
e) **`printf( "%d", j >= i || k == m );`**
f) **`printf( "%d", k + m < j || 3 – j >= k );`**
g) **`printf( "%d", !m );`**
h) **`printf( "%d", !( j – m ) );`**
i) **`printf( "%d", !( k > m ) );`**
j) **`printf( "%d", !( j > k ) );`**

**4.25** Imprima una tabla con los equivalentes en decimal, binario, octal, y hexadecimal. Si desea intentar este ejercicio y no conoce estos sistemas de numeración, primero lea el Apéndice E.

**4.26** Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

**4.27** (*Triples Pitagóricos.*) Un triángulo recto puede tener todos sus lados enteros. Al conjunto de tres valores enteros para los lados del triángulo se le llama Triple Pitagórico. Estos tres lados deben satisfacer la relación que indica que la suma de los cuadrados de los lados es igual al cuadrado de la hipotenusa. Encuentre todos los Triples Pitagóricos para lado1, lado2 y la hipotenusa que no sean mayores que 500. Utilice un triple **`for`** anidado que intente todas las posibilidades. Éste es un ejemplo de computación de "fuerza bruta". No es muy estético para mucha gente. Pero existen muchas razones por las cuales esta técnica es importante. Primero, con el fenomenal incremento en el poder de las computadoras, las soluciones que hubieran tardado años o incluso siglos con la tecnología de hace tan sólo un par de años, ahora puede realizarse en horas, minutos o incluso segundos. Los recientes chips con microprocesadores pueden procesar ¡mil millones de instrucciones por segundo! Segundo, como aprenderá en cursos de computación más avanzados, existe un gran número de problemas interesantes para los cuales no se conocen un método o algoritmo conocido que no sea el de la fuerza bruta. En este libro, investigamos muchos tipos de técnicas para resolver problemas. Aplicaremos muchos métodos de fuerza bruta para distintos problemas interesantes.

**4.28** Una empresa paga a sus empleados como gerentes (quienes reciben un salario semanal fijo), a los empleados por hora (quienes reciben una paga fija por las primeras 40 horas trabajadas, y "hora y media" por las horas extras trabajadas, es decir, 1.5 veces su salario por hora), a los empleados por comisión (quienes reciben $250 más 5.7% de sus ventas brutas semanales), a los empleados por destajo (quienes reciben un monto fijo de dinero por cada elemento que producen, cada empleado por destajo en la empresa trabaja sólo en un tipo de pieza). Escriba un programa que calcule el pago semanal de cada uno de los empleados. Usted no sabe de antemano el número total de empleados. Cada tipo de empleado tiene su propio código de pago: los administradores tienen el código de pago 1, los empleados por hora tienen el código 2, los empleados por comisión tienen el código 3 y los empleados por destajo tienen el código 4. Utilice un **`switch`** para calcular el pago de cada empleado, de acuerdo con su código de empleado. Junto con **`switch`**, indique al usuario (es decir, a la plantilla de empleados) que introduzca los datos que su programa necesita para calcular el pago de cada empleado, de acuerdo con su código de pago.

**4.29** (*Leyes de De Morgan.*) En este capítulo explicamos los operadores lógicos **&&**, **||**, y **!**. Algunas veces, las leyes de De Morgan hacen más conveniente para nosotros el uso de expresiones lógicas. Estas leyes establecen que la expresión !(*condicion1* **&&** *condicion2*) es lógicamente equivalente a la expresión (!*condicion1* **||** !*condicion2*). Utilice las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones lógicas, y después escriba un programa que muestre que en cada caso, tanto la expresión original como la nueva expresión son equivalentes.

a) `!( x < 5 ) && !( y >= 7 )`

b) `!( a == b ) || !( g != 5 )`

c) `!( ( x <= 8 ) && !( y > 4 ) )`

d) `!( ( i > 4 ) || ( j <= 6 )`

**4.30** Rescriba el programa de la figura 4.7 y remplace la instrucción **switch** con una instrucción **if…else** anidada; sea cuidadoso al manejar el caso **default**. Después, rescriba esta nueva versión reemplazando la instrucción anidada **if…else** con una serie de instrucciones **if**; aquí también tenga cuidado al manejar el caso **default** (esto es más difícil que la versión con **if…else** anidado). Este ejercicio demuestra que **switch** es conveniente y que cualquier instrucción **switch** se puede escribir únicamente con instrucciones de selección simple.

**4.31** Escriba un programa que imprima la siguiente figura de rombo. Usted puede utilizar instrucciones **printf** que impriman ya sea un asterisco individual (*****), o un espacio en blanco individual. Maximice el uso de las repeticiones (mediante instrucciones **for** anidadas) y minimice el número de instrucciones **printf**.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

**4.32** Modifique el programa que escribió en el ejercicio 4.31 para que lea un número non en el rango de 1 a 19 para especificar el número de líneas del rombo. Su programa debe desplegar un rombo del tamaño apropiado.

**4.33** Escriba un programa que imprima una tabla de todos lo números romanos equivalentes a los números decimales en el rango de 1 a 100.

**4.34** Escriba un programa que imprima una tabla que contenga los equivalentes de los números 1 a 256 en decimal, binario, octal, y hexadecimal. Si desea intentar este ejercicio y no conoce estos sistemas de numeración, primero lea el Apéndice E.

**4.35** Describa el proceso que utilizaría para remplazar un **do…while** con un **while** equivalente. ¿Qué problema ocurre cuando intenta remplazar un ciclo **while** con un ciclo **do…while**? Suponga que le dicen que tiene que eliminar un ciclo **while** y remplazarlo con un **do…while**. ¿Qué instrucciones de control adicionales necesitaría utilizar, y cómo las utilizaría para garantizar que el resultado del programa sería idéntico al original?

**4.36** Escriba un programa que introduzca un año en el rango de 1994 a 1999, y utilice un ciclo **for** para producir un calendario condensado y claro. Cuidado con los cambios de año.

**4.37** Una crítica de las instrucciones **break** y **continue** es que no son estructuradas En realidad, las instrucciones **break** y **continue** siempre se pueden remplazar con instrucciones estructuradas, sin embargo, hacerlo puede resultar perjudicial. En general, describa cómo eliminaría cualquier instrucción **break** de un ciclo, y cómo la remplazaría con algún equivalente estructurado. [*Pista:* La instrucción **break** abandona un ciclo desde el cuerpo mismo del ciclo. La otra manera de abandonar el ciclo es al fallar la condición de terminación de éste. Considere utilizar una prueba de continuación de ciclo como una segunda prueba que indique un "abandono temprano debido a una condición **break**".] Utilice la técnica que desarrolló aquí, para eliminar la instrucción **break** del programa de la figura 4.11.

**4.38** ¿Qué hace el siguiente programa?

```
for ( i = 1; i <= 5; i++ ) {
   for ( j = 1; j <= 3; j++ ) {
      for ( k = 1; k <= 5; k++ ) {
         printf( “*” );
      printf( “\n” );
   }
   printf( “\n” );
}
```

**4.39** Describa en general cómo eliminaría cualquier instrucción **continue** de un ciclo, y cómo la remplazaría con alguna estructura equivalente. Utilice la técnica que desarrolló aquí, para eliminar la instrucción **continue** del programa de la figura 4.12.



# Funciones en C

## Objetivos

- Comprender cómo construir programas de manera modular mediante pequeñas piezas llamadas funciones.
- Presentar al lector las funciones matemáticas disponibles en la biblioteca estándar de C.
- Crear nuevas funciones.
- Comprender el mecanismo utilizado para pasar información entre funciones.
- Introducir las técnicas de simulación mediante la generación de números aleatorios.
- Comprender cómo escribir y utilizar funciones que se invocan a sí mismas.




*La forma siempre sigue a la función.*
Louis Henri Sullivan

*E pluribus unum.*
(*Uno compuesto por muchos*)
Virgilio

*¡Oh! volvió a llamar ayer, ofreciéndome volver.*
William Shakespeare
*Ricardo II*

*Llámame Ismael.*
Herman Melville
*Moby Dick*

*Cuando me llames así, sonríe.*
Owen Wister

## Plan general



## 5.1 Introducción

La mayoría de los programas de cómputo que resuelven problemas reales son mucho más grandes que los programas que presentamos en el primer capítulo. La experiencia nos ha mostrado que la mejor manera de desarrollar y mantener un programa grande es construirlo a partir de piezas pequeñas o *módulos*, los cuales son más manejables que el programa original. Esta técnica se denomina *divide y vencerás*. En este capítulo describimos las características del lenguaje C que facilitan el diseño, la implementación, la operación y el mantenimiento de programas grandes.

## 5.2 Módulos de programa en C

A los módulos en C se les llama *funciones*. Por lo general, los programas en C se escriben combinando nuevas funciones que escribe el programador con funciones "preempacadas" disponibles en la *biblioteca estándar de C*. En este capítulo explicaremos ambos tipos de funciones. La biblioteca estándar de C proporciona una rica colección de funciones para realizar cálculos matemáticos comunes, manipulación de cadenas, manipulación de caracteres, entrada/salida, y muchas otras operaciones útiles. Esto hace que el trabajo de programador sea más fácil, debido a que estas funciones proporcionan muchas de las capacidades que los programadores necesitan.



**Buena práctica de programación 5.1**

*Conozca la rica colección de funciones de la biblioteca estándar de C.*



**Observación de ingeniería de software 5.1**

*Evite "reinventar la rueda". Cuando sea posible, utilice las funciones de la biblioteca estándar de C, en lugar de escribir nuevas funciones. Esto puede reducir el tiempo de desarrollo de un programa.*



**Tip de portabilidad 5.1**

*Utilizar funciones de la biblioteca estándar de C hace que los programas sean más portables.*

Aunque las funciones de la biblioteca estándar técnicamente no son parte del lenguaje C, invariablemente son proporcionadas con los sistemas de C. Las funciones **`printf`**, **`scanf`** y **`pow`** que utilizamos en los capítulos previos son funciones de la biblioteca estándar.

El programador puede escribir funciones para definir tareas específicas que se podrían utilizar en muchos puntos del programa. A éstas se les llama funciones *definidas por el programador*. Las instrucciones reales que definen a las funciones se escriben solamente una vez, y están ocultas a las demás funciones.

Las funciones se *invocan* mediante una *llamada a función*, la cual especifica el nombre de la función y proporciona información (como *argumentos*) que la función invocada necesita para llevar a cabo su tarea. Una analogía común para esto es la forma jerárquica de administración. Un jefe (la *función que hace la llamada* o *la llamada a función*) le pide a un empleado (la *función invocada*) que realice una tarea y le reporte cuando ésta haya terminado (figura 5.1). Por ejemplo, una función que debe desplegar información en la pantalla llama a la función trabajadora **`printf`** para realizar la tarea; después, **`printf`** despliega la información y la reporta (o la devuelve) a la función que hace la llamada cuando se llevó a cabo la tarea. La función jefe no sabe cómo realiza su tarea la función trabajadora. La función trabajadora podría llamar a otras funciones trabajadoras, y el jefe no se dará cuenta de esto. Muy pronto veremos cómo estos detalles de "ocultamiento" de información promueven la buena ingeniería de software. La figura 5.1 muestra a la función **`jefe`** comunicándose con varias funciones trabajadoras de una manera jerárquica. Observe que **`trabajadora1`** actúa como la función jefe de **`trabajadora4`** y **`trabajadora5`**. Las relaciones entre funciones pueden ser diferentes de la estructura jerárquica que mostramos en la figura.

## 5.3 Funciones matemáticas de la biblioteca

Las funciones matemáticas de la biblioteca permiten al programador realizar ciertos cálculos matemáticos comunes. Aquí utilizamos varias funciones matemáticas para introducir el concepto de funciones. Más adelante, explicaremos muchas de las demás funciones de la biblioteca estándar de C.

Por lo general, las funciones se utilizan en un programa escribiendo el nombre de la función seguido por un paréntesis izquierdo y por el *argumento* (o una lista de argumentos separada por comas) de la función y por el paréntesis derecho. Por ejemplo, un programador que quiere calcular e imprimir la raíz cuadrada de **`900.0`** podría escribir

```
printf( "%.2f", sqrt( 900.0 ) );
```

Cuando se ejecuta esta instrucción, se llama a la función **`sqrt`** de la biblioteca estándar para que calcule la raíz cuadrada del número contenido entre los paréntesis **`(900.0)`**. El número **`900.0`** es el argumento de la función **`sqrt`**. La instrucción anterior imprimirá **`30.00`**. La función **`sqrt`** toma un argumento de tipo **`double`** y devuelve un resultado de tipo **`double`**. Todas las funciones matemáticas de la biblioteca devuelven tipos de datos **`double`**. Observe que los valores **`double`**, como los valores **`float`**, se pueden mostrar utilizando el especificador de conversión **`%f`**.

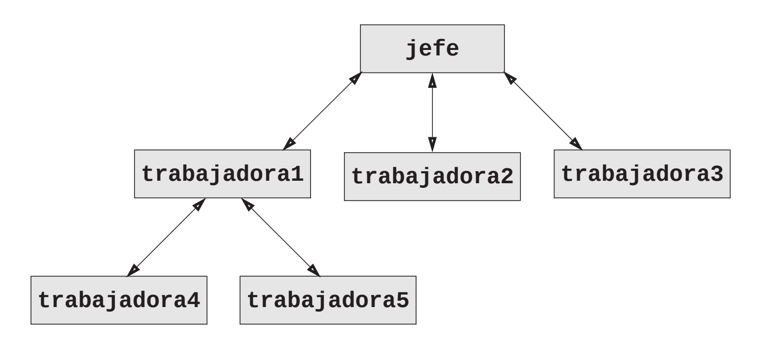



**Figura 5.1** Relación jerárquica entre funciones jefe y funciones trabajadoras.

**Tip para prevenir errores 5.1**

*Cuando utilice las funciones matemáticas de la biblioteca, incluya el encabezado math por medio de la directiva de preprocesador* **#include <math.h>**.

Los argumentos de la función pueden ser constantes, variables, o expresiones. Si **c1=13.0, d=3.0** y **f=4.0**, entonces la instrucción

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

calcula e imprime la raíz cuadrada de **13.0 + 3.0 * 4.0 = 25.0**, a saber, **5.00**.

En la figura 5.2 aparecen algunas funciones matemáticas de la biblioteca de C. En la figura, las variables **x** y **y** son de tipo **double**.

## 5.4 Funciones

Las funciones permiten a los usuarios dividir un programa en módulos. Todas las variables que se definen en una función son *variables locales*, es decir, se conocen sólo en la función en la que se definen. La mayoría de

| Función | Descripción | Ejemplo |
|---|---|---|
| **sqrt( x )** | la raíz cuadrada de *x* | **sqrt( 900.0 )** es **30.0**<br>**sqrt( 9.0)** es **3.0** |
| **exp( x )** | función exponencial $e^x$ | **exp( 1.0 )** es **2.718282**<br>**exp( 2.0 )** es **7.389056** |
| **log( x )** | logaritmo natural de *x* (base *e*) | **log( 2.718282 )** es **1.0**<br>**log( 7.389056 )** es **2.0** |
| **log10 ( x )** | logaritmo de *x* (base 10) | **log10( 1.0 )** es **0.0**<br>**log10( 10.0 )** es **1.0**<br>**log10( 100.0 )** es **2.0** |
| **fabs( x )** | valor absoluto de *x* | **fabs( 5.0 )** es **5.0**<br>**fabs( 0.0 )** es **0.0**<br>**fabs( -5.0 )** es **5.0** |
| **ceil( x )** | redondea *x* al entero más pequeño no menor que *x* | **ceil( 9.2 )** es **10.0**<br>**ceil( -9.8 )** es **-9.0** |
| **floor( x )** | redondea *x* al entero más grande no mayor que *x* | **floor( 9.2 )** es **9.0**<br>**floor( -9.8 )** es **-10.0** |
| **pow( x, y)** | *x* elevada a la potencia *y* ($x^y$) | **pow( 2, 7 )** es **128.0**<br>**pow( 9, 5 )** es **3.0** |
| **fmod ( x, y )** | residuo de *x/y* como un número de punto flotante | **fmod( 13.657, 2.333 )** es **1.992** |
| **sin( x )** | seno trigonométrico de *x* (*x* en radianes) | **sin( 0.0 )** es **0.0** |
| **cos( x )** | coseno trigonométrico de *x* (*x* en radianes) | **cos( 0.0 )** es **1.0** |
| **tan( x )** | tangente trigonométrica de *x* (*x* en radianes) | **tan( 0.0 )** es **0.0** |

**Figura 5.2** Funciones matemáticas comunes de la biblioteca.

las funciones tiene una *lista de parámetros*. Los parámetros proporcionan los medios para transferir información entre funciones. Los parámetros de una función también son variables locales de dicha función.



### Observación de ingeniería de software 5.2

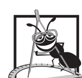
*En los programas que contienen muchas funciones, a menudo* **`main`** *se implementa como un grupo de llamadas a funciones que realizan el grueso del trabajo del programa.*

Existen muchos motivos para “funcionalizar” un programa. El método de divide y vencerás hace que el desarrollo de programas sea más manejable. Otro motivo es la *reutilización de software:* utilizar funciones existentes como bloques de construcción para crear nuevos programas. La reutilización de software es un factor de gran importancia en el movimiento de la programación orientada a objetos, del que usted aprenderá más cuando expliquemos los lenguajes derivados de C, tales como C++, Java y C# (que se pronuncia “C sharp”). Por medio de una buena nomenclatura y una buena definición de funciones, los programas pueden crearse a partir de funciones estándares que cumplan con tareas específicas, en lugar de hacerlo a través de la personalización de código. Esta técnica se conoce como *abstracción*. Utilizamos la abstracción cada vez que escribimos programas que incluyen funciones de la biblioteca como **`printf`**, **`scanf`**, y **`pow`**. Un tercer motivo es el de evitar la repetición de código en un programa. Empacar el código como una función permite que el código se ejecute desde distintas ubicaciones de un programa, simplemente llamando a la función.



### Observación de ingeniería de software 5.3

*Cada función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar de manera clara dicha tarea. Esto facilita la abstracción y promueve la reutilización de software.*



### Observación de ingeniería de software 5.4

*Si usted no puede elegir un nombre conciso que exprese lo que hace la función, es posible que su función intente realizar demasiadas tareas. Por lo general, es mejor dividir dicha función en varias funciones más pequeñas.*

## 5.5 Definición de funciones

Cada programa que presentamos consiste en una función llamada **`main`** que a su vez llama a funciones de la biblioteca estándar para llevar a cabo sus tareas. Ahora explicaremos la manera en que los programadores escriben sus propias funciones personalizadas.

Considere un programa que utiliza una función llamada **`cuadrado`** para calcular e imprimir el cuadrado de los enteros entre 1 y 10 (figura 5.3).

```
/* Figura 5.3: fig05_03.c
   Creación y uso de una función definida por el usuario */
   #include <stdio.h>

int cuadrado( int y ); /* prototipo de la función */

/* la función main comienza la ejecución del programa */
int main()
{
   int x; /* contador */

   /* repite 10 veces el ciclo y calcula e imprime el cuadrado de x */
   for ( x = 1; x <= 10; x++ ) {
      printf( “%d  “, cuadrado( x ) ); /* llamada a la función */
   } /* fin de for */

   printf( “\n” );

   return 0; /* indica terminación exitosa */
```

**Figura 5.3** Uso de una función definida por el programador. (Parte 1 de 2.)

```
20
21 } /* fin de main */
22
23 /* definición de la función cuadrado, devuelve el cuadrado del parámetro */
24 int cuadrado( int y ) /* y es una copia del argumento para la función */
25 {
26    return y * y; /* devuelve el cuadrado de y como un int */
27
28 } /* fin de la función cuadrado */
```

```
1  4  9  16  25  36  49  64  81  100
```

**Figura 5.3** Uso de una función definida por el programador. (Parte 2 de 2.)



### Buena práctica de programación 5.2

*Coloque una línea en blanco entre las definiciones de las funciones para separarlas y mejorar la legibilidad del programa.*

La función **`cuadrado`** se *invoca* o se *llama* en **`main`**, por medio de la instrucción **`printf`** (línea 14)

```
printf( "%d ", cuadrado( x ) ); /* llamada a la función */
```

La función **`cuadrado`** recibe una copia del valor de ***`x`*** en el *parámetro* **`y`** (línea 24). Después, **`cuadrado`** calcula **`y * y`** (línea 26). El resultado regresa a la función **`printf`** en **`main`**, en donde se invocó, y **`printf`** despliega el resultado. Este proceso se repite diez veces por medio de la instrucción de repetición **`for`**.

La definición de la función **`cuadrado`** muestra que ésta espera un parámetro entero **`y`**. La palabra reservada **`int`** que precede al nombre de la función (línea 24) indica que **`cuadrado`** devuelve un resultado entero. La instrucción **`return`** que se encuentra dentro de **`cuadrado`** pasa el resultado del cálculo de vuelta a la llamada de la función.

La línea 5

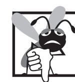
```
int cuadrado( int y ); /* prototipo de la función */
```

es un *prototipo de función*. El **`int`** dentro del paréntesis informa al compilador que **`cuadrado`** espera recibir un valor entero desde la llamada de la función. El **`int`** a la izquierda del nombre de la función **`cuadrado`** informa al compilador que la función **`cuadrado`** devuelve un resultado entero a la llamada de la función. El compilador toma como referencia al prototipo de la función para verificar que las llamadas a **`cuadrado`** (línea 14) contengan el tipo correcto de retorno, el número correcto de argumentos, los tipos correctos de argumentos, y que los argumentos estén en el orden correcto. En la sección 5.6, explicaremos con detalle los prototipos de las funciones.

El formato de una definición de función es:

*tipo-valor-retorno nombre-función*( *lista-parámetros* )
{
*definiciones*
*instrucciones*
}

El *nombre-función* es cualquier identificador válido. El *tipo-valor-retorno* es el tipo de dato del resultado que se devuelve a la llamada de la función. El *tipo-valor-retorno* **`void`** indica que una función no retorna un valor. El compilador asume que un *tipo-valor-retorno* no especificado devuelve un **`int`**. Sin embargo, omitir el tipo de retorno es incorrecto. Juntos, a *tipo-valor-retorno*, *nombre-función*, y *lista-parámetros*, se les conoce como *encabezado de la función*.



### Error común de programación 5.1

*Omitir tipo-valor-retorno en una definición de función es un error de sintaxis si el prototipo de la función especifica un tipo diferente a* ***`int`***.



**Error común de programación 5.2**

*Olvidar devolver un valor desde la función cuando se supone que se debe retornar alguno, puede provocar errores inesperados. El C estándar establece que el resultado de esta omisión es indefinido.*



**Error común de programación 5.3**

*Devolver un valor desde una función, con un tipo de retorno **`void`**, es un error de sintaxis.*



**Buena práctica de programación 5.3**

*Aun cuando un tipo de retorno omitido devuelve de manera predeterminada un **`int`**, siempre establezca el tipo de retorno de manera explícita.*

La *lista-parámetros* es una lista separada por comas que especifican los parámetros recibidos por la función cuando ésta es invocada. Si la función no recibe valor alguno, *lista-parámetros* es **`void`**. Se debe indicar de manera explícita un tipo para cada parámetro, a menos que el parámetro sea de tipo **`int`**. Si no se especifica un tipo, de manera predeterminada se asume el tipo **`int`**.



**Error común de programación 5.4**

*Especificar los parámetros de la función del mismo tipo como **`double x, y`**, en lugar de hacerlo como **`double x, double y`**, podría provocar errores en sus programas. La declaración de parámetros como **`double x, y`**, en realidad hará que **`y`** sea un parámetro de tipo **`int`**, ya que **`int`** es el tipo predeterminado.*



**Error común de programación 5.5**

*Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de la definición de una función, es un error de sintaxis.*



**Error común de programación 5.6**

*Definir otra vez un parámetro de función como una variable local dentro de la función, es un error de sintaxis.*



**Buena práctica de programación 5.4**

*Incluya el tipo de cada parámetro en la lista de parámetros, incluso si el parámetro es del tipo predeterminado **`int`**.*



**Buena práctica de programación 5.5**

*Aunque no es incorrecto hacerlo, en la definición de la función no utilice el mismo nombre para los argumentos que se pasan a una función y para sus parámetros correspondientes. Esto ayuda a evitar la ambigüedad.*

Las *definiciones* e *instrucciones* que se encuentran dentro de las llaves forman el *cuerpo de la función*. Al cuerpo de la función también se le llama *bloque*. Las variables pueden declararse en cualquier bloque, y los bloques pueden anidarse. *Una función no puede definirse dentro de otra función, bajo ninguna circunstancia*.



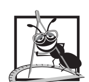
**Error común de programación 5.7**

*Definir una función dentro de otra, es un error de sintaxis.*



**Buena práctica de programación 5.6**

*Elegir nombres significativos de funciones y de parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.*



**Observación de ingeniería de software 5.5**

*Una función no debe ser más grande que una página. Mejor aún, una función no debe ser más grande que la mitad de una página. Las funciones pequeñas promueven la reutilización de software.*



**Observación de ingeniería de software 5.6**

*Los programas deben escribirse como colecciones de funciones pequeñas. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.*



**Observación de ingeniería de software 5.7**

*Una función que tiene un gran número de parámetros podría realizar demasiadas tareas. Considere el dividirla en funciones más pequeñas para realizar tareas separadas. El encabezado de la función debe caber, si es posible, en una sola línea.*

### Observación de ingeniería de software 5.8

*El prototipo de una función, el encabezado de la función y las llamadas a la función deben concordar en número, tipo, orden de argumentos y parámetros, y en el tipo del valor de retorno.*

Existen tres formas de devolver el control de una función invocada al punto en el que se invocó a la función. Si la función no devuelve un resultado, el control simplemente regresa cuando se alcanza la llave derecha de terminación de la función, o cuando se ejecuta la instrucción

**`return;`**

Si la función no devuelve un resultado, la instrucción

**`return`** *expresión*;

devuelve el valor de *expresión* a la llamada de la función.

Nuestro segundo ejemplo utiliza una función definida por el programador llamada **`maximo`**, para determinar y devolver el más grande de tres enteros (figura 5.4). Los tres enteros se introducen mediante **`scanf`** (línea 15). A continuación, los enteros se pasan a **`maximo`** (línea 19), la cual determina el entero más grande. Este valor regresa a **`main`** mediante la instrucción **`return`** de **`maximo`** (línea 39). El valor de retorno se imprime en la instrucción **`printf`** (línea 19).

```
1  /* Figura 5.4: fig05_04.c
2     Encuentra el máximo de tres enteros */
3  #include <stdio.h>
4
5  int maximo( int x, int y, int z ); /* prototipo de la función */
6
7  /* la función main comienza la ejecución del programa */
8  int main()
9  {
10    int numero1; /* primer entero */
11    int numero2; /* segundo entero */
12    int numero3; /* tercer entero */
13
14    printf( “Introduzca tres enteros: “ );
15    scanf( “%d%d%d”, &numero1, &numero2, &numero3 );
16
17    /* numero1, numero2 y numero3 son argumentos
18       para la llamada a la función maximo */
19    printf( “El maximo es: %d\n”, maximo( numero1, numero2, numero3 ) );
20
21    return 0; /* indica terminación exitosa */
22
23 } /* fin de main */
24
25 /* Definición de la función maximo */
26 /* x, y, y z son parámetros */
27 int maximo( int x, int y, int z )
28 {
29    int max = x;      /* asume que x es el mayor */
30
31    if ( y > max ) { /* si y es mayor que max, asigna y a max */
32       max = y;
33    } /* fin de if */
34
```

**Figura 5.4** Función **`maximo`** definida por el programador. (Parte 1 de 2.)

```
35    if ( z > max ) { /* si z es mayor que max, asigna z a max */
36       max = z;
37    } /* fin de if */
38
39    return max;      /* max es el valor más grande */
40
41 } /* fin de la función maximo */
```

```
Introduzca tres enteros: 22 85 17
El maximo es: 85
```

```
Introduzca tres enteros: 85 22 17
El maximo es: 85
```

```
Introduzca tres enteros: 22 17 85
El maximo es: 85
```

**Figura 5.4** Función **maximo** definida por el programador. (Parte 2 de 2.)

## 5.6 Prototipos de funciones

Una de las características más importantes de C es el *prototipo de la función*. Esta característica, la cual fue ideada por los desarrolladores de C++, fue tomada a préstamo por el comité del estándar de C. El prototipo de una función le indica al compilador el tipo de dato devuelto por la función, el número de parámetros que la función espera recibir, los tipos de parámetros, y el orden en el que se esperan dichos parámetros. El compilador utiliza los prototipos de funciones para validar las llamadas a éstas. Las versiones previas de C no realizaban esta clase de verificaciones, por lo que era posible llamar inadecuadamente a las funciones sin que el compilador detectara los errores. Dichas llamadas podían provocar errores fatales en tiempo de ejecución o errores no fatales que provocaban errores lógicos sutiles, pero difíciles de detectar. Los prototipos de las funciones corrigen esta deficiencia.



### Buena práctica de programación 5.7

*Incluya los prototipos de todas las funciones, para aprovechar las capacidades de verificación de tipos de C. Utilice la directiva de preprocesador* **#include** *para obtener los prototipos de función correspondientes a las funciones de la biblioteca estándar, a partir de los encabezados en las bibliotecas apropiadas, o para obtener encabezados que contengan prototipos de funciones desarrolladas por usted y/o sus compañeros de grupo.*

El prototipo de la función **maximo** de la figura 5.4 (línea 5) es

```
int maximo( int x, int y, int z ); /* prototipo de la función */
```

Este prototipo de función establece que **maximo** toma tres argumentos de tipo **int**, y devuelve un resultado de tipo **int**. Observe que el prototipo de la función es el mismo que la primera línea de la definición de la función **maximo**.



### Buena práctica de programación 5.8

*En ocasiones, para efectos de documentación, los nombres de parámetros se incluyen en los prototipos de las funciones (así lo preferimos nosotros). El compilador ignora estos nombres.*



### Error común de programación 5.8

*Olvidar el punto y coma al final del prototipo de la función es un error de sintaxis.*

Una llamada a función que no coincide con el prototipo de la función provoca un error de sintaxis. También se genera un error si el prototipo de la función y la definición de la función no concuerdan. Por ejemplo, si en la figura 5.4 el prototipo de la función se escribiera

```
void maximo( int x, int y, int z );
```

el compilador generaría un error, debido a que el tipo de retorno **void** del prototipo de la función difiere del tipo de retorno **int** del encabezado de la función.

Otra característica importante de los prototipos de funciones es la *coerción de argumentos*, es decir, forzar la conversión de argumentos al tipo apropiado. Por ejemplo, la función matemática **sqrt** de la biblioteca puede llamarse con un argumento entero incluso si el prototipo de la función en **math.h** especifica un argumento **double**, y funcionará correctamente. La instrucción

```
printf( "%.3f\n", sqrt( 4 ) );
```

evalúa de manera correcta **sqrt(4)**, e imprime el valor **2.000**. El prototipo de la función provoca que el compilador convierta el valor entero **4** al valor **double 4.0** antes de que el valor pase a **sqrt**. En general, los valores de argumentos que no corresponden de manera precisa con los tipos de parámetros en el prototipo de la función, se convierten al tipo apropiado antes de que se llame a la función. Estas conversiones pueden provocar resultados incorrectos, si no se siguen las *reglas de promoción de C*. Las reglas de promoción especifican la manera en que los tipos de datos pueden convertirse a otros tipos sin perder datos. En nuestro ejemplo de **sqrt**, un **int** se convierte de manera automática a un **double** sin modificar su valor. Sin embargo, un **double** que se convierte a **int** trunca la parte fraccionaria del valor **double**. Convertir tipos de enteros largos a tipos de enteros cortos (por ejemplo, de **long** a **short**) puede provocar la modificación de los valores.

Las reglas de promoción se aplican de manera automática a expresiones que contienen valores de dos o más tipos de datos (también llamadas *expresiones mixtas*). El tipo de cada valor en una expresión mixta se promueve de manera automática al tipo más "alto" en la expresión (en realidad, se crea una versión temporal de cada valor y se usa para la expresión; los valores originales permanecen sin cambios). La figura 5.5 lista los tipos de datos en orden decreciente con las especificaciones de conversión de tipo para **printf** y **scanf**.

Por lo general, convertir valores a tipos más pequeños provoca la generación de valores incorrectos. Por lo tanto, un valor sólo se puede convertir explícitamente a un tipo más pequeño, asignando el valor a una variable de tipo más pequeño, o mediante un operador de conversión de tipo. Los valores de los argumentos de una función se convierten a los tipos de parámetros en un prototipo de función, como si se asignaran de manera directa a variables de dichos tipos. Si invocamos a nuestra función **cuadrado**, la cual utiliza un parámetro entero (figura 5.3), con un argumento de punto flotante, el argumento se convierte a **int** (un tipo más pequeño) y, en general, cuadrado devolverá un valor incorrecto. Por ejemplo, **cuadrado( 4.5 )** devuelve **16**, no **20.25.**

**Error común de programación 5.9**

*Convertir un tipo de dato de mayor nivel en la jerarquía a uno de menor nivel, puede modificar el valor del dato.*

Si en un programa no se incluye el prototipo de una función, el compilador forma su propio prototipo mediante la primera ocurrencia de la función; ya sea por medio de la definición de la función o de la llamada a ésta. De manera predeterminada, el compilador asume que la función devuelve un **int**, y no asume cosa alguna acerca de los argumentos. Por lo tanto, si los argumentos que se pasan a la función son incorrectos, el compilador no detectará los errores.

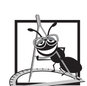
| Tipo de dato | especificación de conversión en printf | especificación de conversión en scanf |
|---|---|---|
| long double | %Lf | %Lf |
| double | %f | %lf |
| float | %f | %f |
| unsigned long int | %lu | %lu |
| long int | %ld | %ld |
| unsigned int | %u | %u |
| int | %d | %d |
| short | %hd | %hd |
| char | %c | %c |

**Figura 5.5** Jerarquía de promoción de los tipos de datos.



**Error común de programación 5.10**

*Olvidar el prototipo de una función provoca un error de sintaxis si en el programa el tipo del valor de retorno no es* **`int`** *y la definición de la función aparece después de la llamada a la función. De lo contrario, olvidar un prototipo de función puede provocar errores en tiempo de ejecución y un resultado inesperado.*



**Observación de ingeniería de software 5.9**

*Un prototipo de función que se coloca fuera de la definición de cualquier función se aplica a todas las llamadas a la función que aparecen después del prototipo de función en el archivo. Un prototipo de función que se coloca en la función se aplica sólo a las llamadas que se hacen en dicha función.*

## 5.7 Encabezados

Cada biblioteca estándar tiene un *encabezado* correspondiente que contiene los prototipos de las funciones de dicha biblioteca y las definiciones de los distintos tipos de datos y constantes necesarios para dichas funciones. La figura 5.6 lista en orden alfabético algunos de los encabezados de la biblioteca estándar que pueden incluirse en los programas. En el capítulo 13, El preprocesador de C, explicaremos con más detalle el término "macros" que utilizamos varias veces en la figura 5.6.

| Encabezado de la biblioteca estándar | Explicación |
|---|---|
| `<assert.h>` | Contiene macros e información para agregar diagnósticos y ayudar en la depuración de programas. |
| `<ctype.h>` | Contiene los prototipos de las funciones que evalúan ciertas propiedades de los caracteres, prototipos de funciones para convertir letras de minúscula a mayúscula y viceversa. |
| `<errno.h>` | Define macros que son útiles para reportar condiciones de error. |
| `<float.h>` | Contiene los límites del sistema con respecto al tamaño de los números de punto flotante. |
| `<limits.h>` | Contiene los límites del sistema con respecto al tamaño de números enteros. |
| `<locale.h>` | Contiene prototipos de funciones e información adicional que permite modificar un programa para adecuarlo al "local" en el que se ejecuta. La idea de "local" permite al sistema de cómputo manipular diferentes convenciones para expresar datos como fechas, horas, montos en moneda y grandes números alrededor del mundo. |
| `<math.h>` | Contiene los prototipos de las funciones matemáticas de la biblioteca. |
| `<setjmp.h>` | Contiene los prototipos de las funciones que permiten evitar la llamada de función usual y la secuencia de retorno. |
| `<signal.h>` | Contiene prototipos de funciones y macros para manipular varias condiciones que se pudieran presentar durante la ejecución del programa. |
| `<stdarg.h>` | Define macros para tratar con una lista de argumentos correspondientes a una función, cuyos números y tipos son desconocidos. |
| `<stddef.h>` | Contiene definiciones comunes de los tipos utilizados por C para realizar ciertos cálculos. |
| `<stdio.h>` | Contiene los prototipos de las funciones de entrada/salida de la biblioteca estándar, y la información que utilizan. |
| `<stdlib.h>` | Contiene los prototipos de las funciones para la conversión de números a texto y de texto a números, asignación de memoria, números aleatorios, y otras funciones de utilidad. |
| `<string.h>` | Contiene los prototipos de las funciones para el procesamiento de cadenas. |
| `<time.h>` | Contiene prototipos de funciones y tipos para manipular la fecha y la hora. |

**Figura 5.6** Algunos de los encabezados de la biblioteca estándar.

El programador puede crear encabezados personalizados. Los encabezados definidos por el programador también deben terminar con **.h**. Un encabezado definido por el programador puede incluirse mediante la directiva del procesador **#include**. Por ejemplo, si el prototipo de nuestra función cuadrado estuviera en el encabezado **cuadrado.h**, podríamos incluir el encabezado al principio del programa mediante la siguiente directiva:

```
#include "cuadrado.h"
```

La sección 13.2 presenta información adicional acerca de la inclusión de encabezados.

## 5.8 Llamada a funciones: Llamada por valor y llamada por referencia

Dos maneras de invocar funciones en muchos lenguajes de programación son la *llamada por valor* y la *llamada por referencia*. Cuando los argumentos se pasan por valor, se crea una *copia* del argumento y se pasa a la función que se invocó. Los cambios hechos a la copia no afectan al valor original de la variable dentro de la función que hace la llamada. Cuando un argumento se pasa por referencia, la función que hace la llamada en realidad permite a la función llamada modificar el valor original de la variable.

La llamada por valor se debe utilizar siempre que las funciones que hacen la llamada no necesiten modificar el valor de la variable original de la llamada. Esto evita los *efectos colaterales* accidentales que afectan de manera importante el desarrollo de sistemas de software correcto y confiable. La llamada por referencia sólo se debe utilizar con funciones confiables que necesiten modificar la variable original.

En C, todas las llamadas se pasan por valor. Como veremos en el capítulo 7, es posible *simular* la llamada por referencia mediante los operadores de dirección e indireccción. En el capítulo 6, veremos que los arreglos se pasan de manera automática mediante una llamada por referencia simulada. Tendremos que esperar hasta el capítulo 7 para que analicemos profundamente este complejo tema. Por ahora, nos concentraremos en la llamada por valor.

## 5.9 Generación de números aleatorios

Ahora echaremos un vistazo breve, pero divertido (espero) a una aplicación de programación popular, a saber, la simulación y los juegos. En ésta y en la siguiente sección, desarrollaremos un programa de juego bien estructurado que incluye múltiples funciones. El programa utiliza muchas de las instrucciones que hemos explicado.

Existe algo especial en los casinos que anima a las personas, desde los jugadores empedernidos que juegan dados en las lujosas mesas de caoba y felpa, hasta las tragamonedas de un cuarto de dólar. Esto especial es el *elemento de azar*, la posibilidad de que la suerte convierta un poco de dinero en un montón de bienestar. El elemento de azar puede introducirse en aplicaciones de cómputo utilizando la función **rand** de biblioteca de C.

Considere la siguiente instrucción:

```
i = rand();
```

La función **rand** genera un entero sin signo entre 0 y **RAND_MAX** (una constante simbólica definida en el encabezado **<stdlib.h>**). El estándar de ANSI, establece que el valor de **RAND_MAX** debe ser al menos de 32767, el cual es el valor máximo para un entero de dos bytes (es decir, 16 bits). Los programas de esta sección se probaron en un sistema C con un valor máximo de 32767 para **RAND_MAX**. Si **rand** realmente genera enteros al azar, cada número entre 0 y **RAND_MAX** tiene la misma *oportunidad (o probabilidad)* de ser elegido cada vez que se invoca a **rand**.

El rango de valores que produce **rand** de manera directa, a menudo difiere del requerido por la aplicación. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "cara" y 1 para "cruz". Un programa que simula el tiro de un dado de seis lados requiere enteros al azar entre 1 y 6.

Para demostrar la función **rand**, desarrollemos un programa que simule 20 tiros de un dado de seis lados y que despliegue el valor de cada tiro. El prototipo de función para la función **rand** se puede encontrar en **<stdlib.h>**. Para producir números en el rango de 1 a 5, utilizamos el operador módulo (**%**) junto con **rand** de la siguiente manera:

```
rand() % 6
```

A esto se le llama *escalamiento*. Al número 6 se le denomina *factor de escalamiento*. Después *cambiamos* el rango de los números que se producen, sumando 1 a nuestro resultado previo. La salida de la figura 5.7 confirma que los resultados se encuentran en el rango de 1 a 6.

```
/* Figura 5.7: fig05_07.c
   Escalamiento y cambio de enteros producidos por 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int i; /* contador */

   /* repite 20 veces */
   for ( i = 1; i <= 20; i++ ) {

      /* obtiene y despliega un número aleatorio entre 1 y 6 */
      printf( “%10d”, 1 + ( rand() % 6 ) );

     /* si el contador es divisible entre 5, comienza una nueva línea de salida */
      if ( i % 5 == 0 ) {
         printf( “\n” );
      } /* fin de if */

   } /* fin de for */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

**Figura 5.7** Escalamiento y cambio de enteros producidos por **1 + rand()%6**.

Para mostrar que los números producidos por la función **rand** ocurren aproximadamente con la misma probabilidad, simulemos 6,000 tiros de dados con el programa de la figura 5.8. Cada entero en el rango de 1 a 6 debe aparecer aproximadamente 1,000 veces.

```
/* Figura 5.8: fig05_08.c
 Tiro de un dado de seis lados 6000 veces */
include <stdio.h>
include <stdlib.h>

/* la función main comienza la ejecución del programa */
main()
{
   int frecuencia1 = 0; /* contador del tiro 1 */
   int frecuencia2 = 0; /* contador del tiro 2 */
   int frecuencia3 = 0; /* contador del tiro 3 */
   int frecuencia4 = 0; /* contador del tiro 4 */
   int frecuencia5 = 0; /* contador del tiro 5 */
   int frecuencia6 = 0; /* contador del tiro 6 */

   int tiro; /* contador de tiros, valores de 1 a 6000 */
```

**Figura 5.8** Tiro de un dado de seis lados 6,000 veces. (Parte 1 de 2.)

```
    int cara; /* representa un tiro del dado, valores de 1 a 6 */

    /* repite 6000 veces y resume los resultados */
    for ( tiro = 1; tiro <= 6000; tiro++ ) {
       cara = 1 + rand() % 6; /* número aleatorio de 1 a 6 */

       /* determina el valor de cara e incrementa el contador apropiado */
       switch ( cara ) {

          case 1: /* tiro 1 */
             ++frecuencia1;
             break;

          case 2: /* tiro 2 */
             ++frecuencia2;
             break;

          case 3: /* tiro 3 */
             ++frecuencia3;
             break;

          case 4: /* tiro 4 */
             ++frecuencia4;
             break;

          case 5: /* tiro 5 */
             ++frecuencia5;
             break;

          case 6: /* tiro 6 */
             ++frecuencia6;
             break; /* opcional */
       } /* fin de switch */

    } /* fin de for */

    /* despliega los resultados en forma tabular */
    printf( “%s%13s\n”, “Cara”, “Frecuencia” );
    printf( “   1%13d\n”, frecuencia1 );
    printf( “   2%13d\n”, frecuencia2 );
    printf( “   3%13d\n”, frecuencia3 );
    printf( “   4%13d\n”, frecuencia4 );
    printf( “   5%13d\n”, frecuencia5 );
    printf( “   6%13d\n”, frecuencia6 );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Cara   Frecuencia
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

**Figura 5.8** Tiro de un dado de seis lados 6,000 veces. (Parte 2 de 2.)

Como muestra la salida del programa, podemos simular el tiro de un dado de seis lados si cambiamos y escalamos los valores producidos por **`rand`**. Observe que el programa nunca debe alcanzar el caso **`default`** proporcionado en la instrucción **`switch`**. Observe también el uso del especificador de conversión **`%s`** para imprimir las cadenas **`"Cara"`** y **`"Frecuencia"`** como encabezados de columnas (línea 54). Después de que estudiemos los arreglos en el capítulo 6, mostraremos cómo reemplazar por completo la instrucción **`switch`** de manera elegante por medio de una instrucción de una sola línea. De nuevo, la ejecución del programa de la figura 5.7 produce la siguiente salida:

```
          6          6          5          5          6
          5          1          1          5          3
          6          6          2          4          2
          6          2          3          4          1
```

Observe que se imprime exactamente la misma secuencia de valores. ¿Cómo pueden ser estos, números aleatorios? Irónicamente esta repetición es una característica importante de la función **`rand`**. Cuando depuramos un programa, esta repetición es esencial para mostrar que las correcciones a un programa funcionan de manera apropiada.

En realidad, la función **`rand`** genera *números pseudoaleatorios*. Al llamar repetidamente a **`rand`**, se produce una secuencia de números que parece ser aleatorios. Sin embargo, la secuencia se repite a sí misma cada vez que se ejecuta el programa. Una vez que depuramos el programa por completo, lo podemos condicionar para producir secuencias diferentes de números aleatorios para cada ejecución. A esto se le denomina *randomizar*, y se lleva a cabo mediante la función **`srand`** de la biblioteca. La función **`srand`** toma un entero **`unsigned`** como argumento y establece la *semilla* de la función **`rand`** para producir una secuencia diferente de números aleatorios para cada ejecución del programa.

En la figura 5.9 mostramos el uso de **`srand`**. En el programa, utilizamos el tipo de dato **`unsigned`**, el cual es una abreviatura de **`unsigned int`**. Un entero se almacena en al menos dos bytes de memoria, y puede contener tanto valores positivos como negativos. Una variable de tipo **`unsigned`** también se almacena en al menos dos bytes de memoria. Un entero de dos bytes **`unsigned int`** sólo puede contener valores positivos en el rango de 0 a 65535. Un entero **`unsigned int`** de cuatro bytes sólo puede contener valores positivos en el rango de 0 a 4294967295. La función **`srand`** toma un valor **`unsigned`** como argumento. El especificador de conversión **`%u`** se utiliza para leer un valor **`unsigned`** por medio de **`scanf`**. El prototipo de la función de **`srand`** se encuentra en **`<stdlib.h>`**.

```
/* Figura 5.9: fig05_09.c
   Randomización del programa de dados */
#include <stdlib.h>
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int i;              /* contador */
   unsigned semilla; /* número que se utiliza para establecer la semilla
   del generador de números aleatorios */

   printf( "Introduzca la semilla: " );
   scanf( "%u", &semilla ); /* observe el %u para un unsigned */

   srand( semilla ); /* establece la semilla del generador de números aleatorios */

   /* repite 10 veces */
   for ( i = 1; i <= 10; i++ ) {
```

**Figura 5.9** Randomización del programa de tiro de dados. (Parte 1 de 2.)

```

      /* obtiene y despliega un número aleatorio entre 1 y 6 */
      printf( "%10d", 1 + ( rand() % 6 ) );

      /* si contador es divisible entre 5, comienza una nueva línea de salida */
      if ( i % 5 == 0 ) {
         printf( "\n" );
      } /* fin de if */

   } /* fin de for */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca la semilla: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Introduzca la semilla: 867
         2         4         6         1         6
         1         1         3         6         2
```

```
Introduzca la semilla: 67
         6         1         4         6         2
         1         6         1         6         4
```

**Figura 5.9** Randomización del programa de tiro de dados. (Parte 2 de 2.)

Ejecutemos el programa varias veces y observemos los resultados. Observe que cada vez que ejecutamos el programa obtenemos una secuencia de números *diferente*, debido a que proporcionamos una semilla diferente.

Si queremos randomizar sin tener que introducir una semilla diferente cada vez, podemos utilizar la siguiente instrucción:

```
srand( time( NULL ) );
```

Esto provoca que la computadora lea su reloj y obtenga el valor para la semilla de manera automática. La función **time** devuelve la hora del día en segundos. Este valor se convierte en un entero sin signo y se utiliza como semilla para la generación de números aleatorios. La función **time** toma un argumento **NULL** (**time** es capaz de proporcionar al programador la cadena que representa la hora del día; **NULL** deshabilita esta capacidad para la llamada específica a la función). El prototipo de la función para **time** se encuentra en **<time.h>**.

Los valores que **rand** produce de manera directa se encuentran en el rango:

```
0 ≤ rand() ≤ RAND_MAX
```

Previamente demostramos la manera de escribir una sencilla instrucción para simular el tiro de un dado de seis lados:

```
Cara = 1 + rand() % 6;
```

Esta instrucción siempre asigna un valor entero (aleatorio) a la variable **cara**, en el rango de 1 ≤ **cara** ≤ 6. Observe que la longitud del rango (es decir, el número de enteros consecutivos en el rango) es de 6 y el número de inicio es 1. Respecto a la instrucción anterior, vemos que el rango se determina por medio del número que utilizamos para escalar **rand** con el operador módulo (es decir, 6), y el número inicial del rango es igual al número (es decir, 1) que se suma a **rand%6**. Podemos generalizar este resultado de la siguiente manera:

```
n = a + rand() % b;
```

en donde **a** es el *valor de cambio* (el cual es igual al primer número del rango deseado de enteros consecutivos), y **b** es el factor de escalamiento (que es igual a la longitud del rango deseado de enteros consecutivos). En los ejercicios, veremos que es posible elegir enteros de manera aleatoria a partir de conjuntos de valores diferentes a los rangos consecutivos de enteros.

**Error común de programación 5.11**

*Usar* ***srand*** *en un lugar de* ***rand*** *para generar números aleatorios.*

## 5.10 Ejemplo: Un juego de azar

Uno de los juegos de azar más populares es el juego de dados conocido como “craps”, el cual se juega en casinos y patios traseros alrededor del mundo. Las reglas del juego son simples.

> *El jugador tira dos dados. Cada dado tiene seis caras. Estas caras contienen 1, 2, 3, 4, 5 y 6 puntos. Una vez que los dados caen, se calcula la suma de los puntos que se encuentran en las caras que ven hacia arriba. Si la suma es igual a 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado “craps”), el jugador pierde (es decir, la “casa” gana). Si la suma es 4, 5, 6, 8, 9, o 10 en el primer tiro, entonces la suma se convierte en el “punto” del jugador. Para ganar, usted debe continuar tirando los dados hasta que “haga su punto”. El jugador pierde si tira un 7 antes de hacer su punto.*

La figura 5.10 simula el juego de craps, y la figura 5.11 muestra varias ejecuciones de ejemplo.

```
/* Figura 5.10: fig05_10.c
   Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contiene el prototipo de la función time */

/* constantes de enumeración que representan el estado del juego */
enum Estatus { CONTINUA, GANA, PIERDE };

int tiraDados( void ); /* prototipo de la función */

/* la función main comienza la ejecución del programa */
int main()
{
   int suma;        /* suma del tiro de datos */
   int miPunto;     /* punto ganado */

   enum Estatus estatusJuego /* puede contener CONTINUA, GANA o PIERDE */

   /* randomiza el generador de números aleatorios mediante la función time */
   srand( time( NULL ) );

   suma = tiraDados(); /* primer tiro de los dados */

   /* determina el estado del juego basado en la suma de los dados */
   switch( suma ) {

      /* gana en el primer tiro */
      case 7:
      case 11:
         estatusJuego = GANA;
```

**Figura 5.10** Programa para simular el juego de craps. (Parte 1 de 3.)

```
         break;

      /* pierde en el primer tiro */
      case 2:
      case 3:
      case 12:
         estatusJuego = PIERDE;
         break;

      /* recuerda el punto */
      default:
         estatusJuego = CONTINUA;
         miPunto = suma;
         printf( "Su punto es %d\n", miPunto );
         break; /* opcional */
   } /* fin de switch */

   /* mientras el juego no se complete */
   while ( estatusJuego == CONTINUA ) {
      suma = tiraDados(); /* tira de nuevo los dados */

      /* determina el estatus del juego */
      if ( suma == miPunto ) { /* gana por punto */
         estatusJuego = GANA; /* fin del juego, el jugador gana */
      } /* fin de if */
      else {

         if ( suma == 7 ) { /* pierde al tirar 7 */
            estatusJuego = PIERDE; /* termina el juego, el jugador pierde */
         } /* fin de if */

      } /* fin de else */

   } /* fin de while */

   /* despliega mensaje de triunfo o derrota */
   if ( estatusJuego == GANA ) { /* ¿Ganó el jugador? */
      printf( "El jugador gana \n" );
   } /* fin de if */
   else { /* el jugador pierde */
      printf( "El jugador pierde\n" );
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* tiro de dados, calcula la suma y despliega los resultados */
int tiraDados( void )
{
   int dado1;    /* primer dado */
   int dado2;    /* segundo dado */
   int sumaTemp; /* suma de los dados */
```

**Figura 5.10** Programa para simular el juego de craps. (Parte 2 de 3.)

```
85
86      dado1 = 1 + ( rand() % 6 ); /* toma el aleatorio para el dado1 */
87      dado2 = 1 + ( rand() % 6 ); /* toma el aleatorio para el dado2 */
88      sumaTemp = dado1 + dado2;   /* suma el dado1 y el dado2 */
89
90      /* despliega los resultados de este tiro */
91      printf( "El jugador tiro %d + %d = %d\n", dado1, dado2, sumaTemp );
92
93      return sumaTemp; /* devuelve la suma de los dados */
94
95   } /* fin de la función tiraDados */
```

**Figura 5.10** Programa para simular el juego de craps. (Parte 3 de 3.)

```
El jugador tiro 1 + 5 = 6
Su punto es 6
```

```
El jugador tiro 2 + 1 = 3
El jugador tiro 2 + 6 = 8
El jugador tiro 2 + 3 = 5
El jugador tiro 6 + 2 = 8
El jugador tiro 4 + 2 = 6
El jugador gana
```

```
El jugador tiro 1 + 1 = 2
El jugador pierde
```

```
El jugador tiro 5 + 3 = 8
Su punto es 8
El jugador tiro 2 + 4 = 6
El jugador tiro 5 + 4 = 9
El jugador tiro 5 + 3 = 8
El jugador gana
```

**Figura 5.11** Ejemplo de ejecuciones del juego Craps.

En las reglas del juego, observe que el jugador debe tirar dos dados en el primer tiro, y también lo debe hacer en los demás tiros subsecuentes. Definimos una función llamada **tiraDados** para lanzar los dados y calcular e imprimir la suma. La función **tiraDados** se define una sola vez, pero se invoca desde dos ubicaciones diferentes en el programa (líneas 23 y 51). De manera interesante, **tiraDados** no toma argumentos, así que indicamos **void** dentro de la lista de parámetros (línea 80). La función **tiraDados** devuelve la suma de los dos dados, por lo que indicamos el tipo de retorno **int** en el encabezado de la función.

El juego es razonablemente complicado. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro subsiguiente. La variable **estatusJuego**, definida para que sea de un nuevo tipo **enum Estatus**, almacena el estado actual. La línea 8 crea un tipo definido por el programador llamada *enumeración*. Una enumeración, definida mediante la palabra reservada **enum**, es un conjunto de constantes enteras representadas por medio de identificadores. En ocasiones, a las *constantes de enumeración* se les llama constantes simbólicas; esto es, constantes representadas por medio de símbolos. Los valores en una enumeración comienzan en **0** y se incrementan en **1**. En la línea 8, la constante **CONTINUA** tiene el valor **0**. **GANA** tiene el valor **1** y **PIERDE** tiene el valor **2**. También, en un **enum**, es posible asignar un valor entero a cada identificador (revise el capítulo 13). Los identificadores de una enumeración deben ser únicos, pero los valores pueden estar duplicados.



**Error común de programación 5.12**

*Asignar un valor a una constante de enumeración después de que se define es un error de sintaxis.*



**Buena práctica de programación 5.9**

*Utilice sólo letras mayúsculas en los nombres de las constantes de enumeración para hacer que resalten en el programa, y para indicar que las constantes de enumeración no son variables.*

Cuando se gana el juego, ya sea en el primer tiro o en un tiro subsiguiente, **estatusJuego** se establece en **GANA**. Cuando se pierde el juego, ya sea en el primer tiro o en uno subsiguiente, **estatusJuego** se establece en **PIERDE**. De lo contrario se establece en **CONTINUA**, y el juego continúa.

Después del primer tiro, si el juego termina, se ignora la instrucción **while** (línea 50), debido a que **estatusJuego** no se encuentra en **CONTINUA**. El programa continúa con la instrucción **if…else** de línea 68, la cual imprime "**El jugador gana**" si **estatusJuego** es **GANA**, o de lo contrario "**El jugador pierde**".

Después del primer tiro, si el juego aún no termina, entonces la **suma** se guarda en **miPunto**. La ejecución procede con la instrucción **while** (línea 50), debido a que **estatusJuego** es **CONTINUA**. Cada vez que se ejecuta el **while**, se llama a **tiraDados** para producir una nueva suma. Si la suma coincide con **miPunto**, **estatusJuego** se establece en **GANA** para indicar que el jugador ganó; la prueba del **while** falla; la instrucción **if…else** (línea 68) imprime "**El jugador gana**"; y termina la ejecución. Si **suma** es igual a **7** (línea 59), **estatusJuego** se establece en **PIERDE** para indicar que el jugador perdió; la prueba del **while** falla; la instrucción **if…else** (línea 68) imprime "**El jugador pierde**"; y termina la ejecución.

Observe la interesante arquitectura de control del programa. Utilizamos dos funciones, **main** y **tiraDados** (y las instrucciones **switch**, **while**, y las instrucciones anidadas **if…else** e **if** ). En los ejercicios, investigaremos varias características interesantes relacionadas con este juego.

## 5.11 Clases de almacenamiento

En los capítulos 2 a 4, utilizamos identificadores para los nombres de variables. Los atributos de las variables incluyen el nombre, el tipo, el tamaño y el valor. En este capítulo, además utilizaremos identificadores como nombres de funciones definidas por el usuario. En realidad, en un programa, cada identificador tiene otros atributos que incluyen la *clase de almacenamiento*, la *duración de almacenamiento*, el *alcance* y la *vinculación*.

C proporciona cuatro clases de almacenamiento que se indican por medio de los *especificadores de clase de almacenamiento*: ***auto***, ***register***, ***extern*** y ***static***. La *clase de almacenamiento* de un identificador determina su *duración de almacenamiento*, *alcance* y *vinculación*. La *duración de almacenamiento* de un identificador es el periodo durante el cual, dicho identificador existe en memoria. Algunos identificadores existen de manera breve, algunos se crean y se destruyen repetidamente, y otros existen durante toda la ejecución del programa. El *alcance* de un identificador se refiere al lugar en donde se puede hacer referencia a él en un programa. Se puede hacer referencia a algunos identificadores a través de todo un programa, y a otros sólo en partes del programa. La *vinculación* de un identificador determina (para un programa con múltiples archivos fuente) si éste se reconoce sólo en el archivo fuente actual o en cualquier archivo fuente con las declaraciones apropiadas; este tema lo explicaremos en el capítulo 14. En esta sección explicamos las cuatro clases de almacenamiento y la duración de almacenamiento. En la sección 5.12 explicamos el alcance de los identificadores. En el capítulo 14, Otros temas de C, explicamos la vinculación y la programación con múltiples archivos fuente.

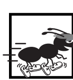
Los cuatro especificadores de clase de almacenamiento pueden clasificarse en dos duraciones de almacenamiento: *duración automática de almacenamiento*, y *duración estática de almacenamiento*. Para declarar variables con duración automática de almacenamiento utilizamos las palabras reservadas **auto** y **register**. Las variables con duración automática de almacenamiento se crean cuando el programa entra al bloque en el que están definidas; éstas existirán mientras el bloque esté activo, y se destruirán cuando el programa abandona el bloque.

Sólo las variables pueden tener una duración automática de almacenamiento. Por lo general, las variables locales de una función (aquellas que se declaran en la lista de parámetros o en el cuerpo de la función) tienen una duración automática de almacenamiento. La palabra reservada **auto** declara de manera explícita variables con duración automática de almacenamiento. Por ejemplo, las siguientes declaraciones indican que las va-

riables **double x** y **y** son variables locales automáticas. Y existen sólo en el cuerpo de la función en la cual aparecen las declaraciones:

```
auto double x, y;
```

De manera predeterminada, las variables locales tienen una duración automática de almacenamiento, de tal modo que la palabra reservada **auto** rara vez se utiliza. Durante el resto del libro, nos referiremos a las variables con duración automática de almacenamiento simplemente como *variables automáticas*.



### Tip de rendimiento 5.1

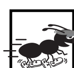
*El almacenamiento automático es una manera de ahorrar memoria, ya que las variables automáticas existen sólo cuando son requeridas. Éstas se crean cuando la función en la que se definen inicia su ejecución, y se destruyen cuando termina su ejecución.*



### Observación de ingeniería de software 5.10

*El almacenamiento automático es un ejemplo del principio del menor privilegio, permite el acceso a los datos sólo cuando es absolutamente necesario. ¿Para que tener variables almacenadas en memoria y accesibles si, de hecho, no son necesarias?*

Por lo general, los datos de un programa que se encuentran en versión de lenguaje máquina se cargan en los registros para cálculos y otros procesos.



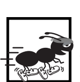
### Tip de rendimiento 5.2

*El especificador* ***register*** *puede colocarse antes de la declaración de la variable automática para sugerir al compilador que mantenga a la variable en uno de los registros de hardware de alta velocidad. Si utiliza de manera intensa variables tales como contadores o totales, éstas pueden mantenerse en registros de hardware; de este modo podrá evitar la sobrecarga producida por pasar las variables de la memoria a los registros y almacenar nuevamente los registros en memoria.*

El compilador podría ignorar las declaraciones **register**. Por ejemplo, podría no haber un número suficiente de registros disponibles para el uso del compilador. La siguiente declaración sugiere que la variable entera **contador** puede colocarse en uno de los registros de la computadora e inicializarla en1:

```
register int contador = 1;
```

La palabra reservada **register** puede utilizarse sólo con variables de duración automática de almacenamiento.



### Tip de rendimiento 5.3

*A menudo, las declaraciones* ***register*** *son innecesarias. Con frecuencia, los compiladores optimizados actuales son capaces de reconocer las variables utilizadas y pueden decidir colocarlas en registros, sin necesidad de que el programador las declare como* ***register****.*

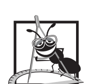
Las palabras reservadas **extern** y **static** se utilizan para declarar identificadores para variables y funciones de duración estática de almacenamiento. Los identificadores de duración estática de almacenamiento existen desde el punto en el que el programa comienza la ejecución. En el caso de las variables, el almacenamiento se asigna y se inicializa una sola vez: cuando comienza la ejecución del programa. Para las funciones, el nombre de la función existe cuando comienza la ejecución del programa. Sin embargo, aun cuando las variables y los nombres de las funciones existan desde el momento de la ejecución del programa, esto no significa que se pueda acceder a los identificadores a lo largo de todo el programa. La duración del almacenamiento y el alcance (en donde se puede utilizar el nombre) son temas aparte que explicaremos en la sección 5.12.

Existen dos tipos de identificadores con duración estática de almacenamiento: los identificadores externos (tales como variables globales y nombres de función) y las variables locales declaradas con el especificador de la clase de almacenamiento **static**. Las variables globales y los nombres de función pertenecen, de manera predeterminada, a la clase de almacenamiento **extern**. Las variables globales se crean al colocar las declaraciones de las variables fuera de cualquier definición de función, y retienen sus valores a lo largo de la ejecución del programa. Es posible hacer referencia a las variables globales y a las funciones por medio de cualquier función que siga sus declaraciones o sus definiciones dentro del archivo. Ésta es una razón para utilizar los prototipos de las funciones; cuando incluimos **stdio.h** en un programa que invoca a **printf**, el prototipo de la función se coloca al inicio de nuestro archivo para que el nombre **printf** sea reconocido en el resto del archivo.



**Observación de ingeniería de software 5.11**

*Definir una variable como global, en lugar de hacerlo como local, permite que ocurran efectos colaterales, por ejemplo, cuando una función que no necesita acceso a la variable la modifica de manera accidental o maliciosa. En general, debe evitarse el uso de variables globales, excepto en ciertas situaciones con requerimientos especiales de rendimiento (como explicaremos en el capítulo 14).*



**Observación de ingeniería de software 5.12**

*Las variables que se utilizan sólo en una función en particular, deben definirse como variables locales en esa función y no como variables externas.*

Las variables locales que se declaran con la palabra reservada **`static`** sólo se reconocen en la función en la que se definen, pero a diferencia de las variables automáticas, las variables locales **`static`** retienen su valor, incluso cuando se sale de la función. La siguiente vez que se invoca a la función, la variable local **`static`** contiene el valor que tenía cuando la función terminó por última vez. La siguiente instrucción declara a la variable local **`cuenta`** como **`static`** y hace que se inicialice en 1.

```
static int cuenta = 1;
```

Todas las variables numéricas de duración estática de almacenamiento se inicializan en cero, si el programador no las inicializa de manera explícita.



**Error común de programación 5.13**

*Utilizar múltiples especificadores de clase de almacenamiento para un identificador. Sólo se puede aplicar un especificador de clase de almacenamiento a un identificador.*

Las palabras reservadas **`extern`** y **`static`** tienen un significado especial cuando se aplican explícitamente a identificadores externos. En el capítulo 14, Otros temas de C, explicamos el uso explícito de **`extern`** y **`static`** con identificadores externos y con programas que tienen múltiples archivos fuente.

## 5.12 Reglas de alcance

El *alcance* de un identificador es la porción del programa en la que se puede hacer referencia a un identificador. Por ejemplo, cuando definimos una variable local en un bloque, sólo se puede hacer referencia a ella en el bloque o en los bloques anidados dentro del mismo bloque. Los cuatro tipos de alcance para un identificador son: *alcance de función*, *alcance de archivo*, *alcance de bloque*, y *alcance de prototipo de función*.

Las etiquetas (un identificador seguido por dos puntos, como en **`inicio:`**) son los únicos identificadores con *alcance de función*. Las etiquetas pueden utilizarse en cualquier parte de la función en la que aparecen, pero no se puede hacer referencia a ellas fuera del cuerpo de la función. Las etiquetas se utilizan en las instrucciones **`switch`** (como etiquetas **`case`**), y en las instrucciones **`goto`** (revise el capítulo 14). Las etiquetas son detalles de implementación que las funciones se ocultan entre sí. Este ocultamiento, formalmente llamado *ocultamiento de información*, es un medio para implementar el *principio del menor privilegio*, uno de los principios fundamentales de la buena ingeniería de software.

Un identificador que se declara fuera de cualquier función tiene *alcance de archivo*. A tal identificador se le "reconoce" (es decir, es accesible) en todas las funciones desde el punto en el que se declara, y hasta el final del archivo. Las variables globales, las definiciones de funciones y los prototipos de funciones que se colocan fuera de una función tienen alcance de archivo.

Los identificadores que se definen dentro de un bloque tienen *alcance de bloque*. El alcance de bloque termina al encontrar la llave derecha (}) de terminación de un bloque. Las variables locales que se encuentran al principio de una función tienen alcance de bloque, así como los parámetros de la función, los cuales se consideran variables locales de la función. Cualquier bloque puede contener definiciones de variables. Cuando los bloques se encuentran anidados, y un identificador en un bloque externo tiene el mismo nombre que un identificador en el bloque interno, el identificador en el bloque externo se "oculta" hasta que el bloque interno termina. Esto significa que durante la ejecución del bloque interno, éste ve el valor de su propio identificador local y no el valor del identificador idéntico del bloque externo. Las variables locales que se declaran como **`static`** también tienen alcance de bloque, aun cuando existan desde el momento en que el programa comienza la ejecución. Además, la duración de almacenamiento no afecta el alcance de un identificador.

Los únicos identificadores con *alcance de prototipo de función* son aquellos que se utilizan en la lista de parámetros de un prototipo de función. Como mencionamos anteriormente, los prototipos de función no requieren nombres dentro de la lista de parámetros; sólo se requieren los tipos. Si se utiliza un nombre en la lista de parámetros del prototipo de una función, el compilador ignora el nombre. Los identificadores que se utilizan en el prototipo de una función pueden reutilizarse en cualquier parte del programa, sin crear ambigüedades.

### Error común de programación 5.14

*Utilizar de manera accidental el mismo nombre para un identificador en un bloque interno y en un bloque externo, cuando de hecho, el programador quiere que el identificador del bloque externo se encuentre activo durante la ejecución del bloque interno.*

### Tip para prevenir errores 5.2



*Evite nombres de variables que oculten nombres con alcances externos. Esto se puede llevar a cabo simplemente evitando el uso de identificadores duplicados en un programa.*

La figura 5.12 muestra cuestiones relacionadas con el alcance de variables globales, variables locales automáticas, y variables locales **`static`**. Definimos una variable global **`x`** y la inicializamos en 1 (línea 9). Esta variable global se encuentra oculta en cualquier bloque (o función) en el que se defina otra variable **`x`**. En **`main`**, definimos una variable local **`x`** y la inicializamos en 5 (línea 14). Después, esta variable se imprime para mostrar que la variable global **`x`** se oculta en **`main`**. A continuación, definimos un bloque dentro de **`main`** con otra variable local **`x`** que inicializamos en 7 (línea 19). Esta variable se imprime para mostrar que **`x`** se oculta en el bloque externo de **`main`**. La variable **`x`** se destruye de manera automática cuando salimos del bloque, y se imprime de nuevo la variable local **`x`** en el bloque externo de **`main`**, para mostrar que ya no está oculta. El programa define tres funciones que no toman argumentos y no devuelven valor alguno. La función **`usoLocal`** define una variable automática **`x`**, y la inicializa en 25 (línea 42). Cuando se invoca a la función **`usoLocal`**, la variable se imprime, se incrementa, y se vuelve a imprimir antes de salir de la función. Cada vez que se llama a la función, la variable automática **`x`** se inicializa en 25. En la función **`usoStaticLocal`** definimos a la variable **`static x`** y la inicializamos en 50 (línea 55). Las variables declaradas como locales retienen su valor aun cuando se encuentran fuera de alcance. Cuando se invoca a **`usoStaticLocal`**, se imprime **`x`**, se incrementa, y se vuelve a imprimir antes de salir de la función. En la siguiente llamada a esta función, la variable **`static x`** contendrá el valor 51. La función **`usoGlobal`** no define variable alguna. Por lo tanto, cuando hace referencia a la variable **`x`**, utiliza la variable global **`x`** (línea 9). Cuando se llama a **`usoGlobal`**, se imprime la variable global, se multiplica por 10, y se imprime antes de salir de la función. La siguiente vez que se llama a la función **`usoGlobal`**, la variable global contiene el valor modificado, 10. Por último, el programa imprime de nuevo la variable local **`x`** en **`main`** (línea 33), para mostrar que ninguna de las llamadas a las funciones modificó el valor de **`x`**, ya que todas las funciones hacen referencia a variables con otros alcances.

```
 1  /* Figura 5.12: fig05_12.c
 2     Ejemplo de alcance  */
 3  #include <stdio.h>
 4
 5  void usoLocal( void );        /* prototipo de función */
 6  void usoStaticLocal( void );  /* prototipo de función */
 7  void usoGlobal( void );       /* prototipo de función */
 8
 9  int x = 1; /* variable global */
10
11  /* la función main comienza la ejecución del programa */
12  int main()
13  {
14     int x = 5; /* variable local en main */
15
```

**Figura 5.12** Ejemplo de alcance. (Parte 1 de 3.)

```
   printf("la x local fuera del alcance de main es %d\n", x );

   { /* comienza el nuevo alcance */
      int x = 7; /* variable local con nuevo alcance */

      printf( "la x local en el alcance interno de main es %d\n", x );
   } /* fin de nuevo alcance */

   printf( "la x local en el alcance externo de main es %d\n", x );

   usoLocal();       /* usoLocal contiene una x local */
   usoStaticLocal(); /* usoStaticLocal contiene una x local estática */
   usoGlobal();      /* usoGlobal utiliza una x global */
   usoLocal();       /* usoLocal reinicializa la x local automática */
   usoStaticLocal(); /* static local x retiene su valor previo */
   usoGlobal();      /* x global también retiene su valor */

   printf( "\nx local en main es %d\n", x );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* usoLocal reinicializa a la variable local x durante cada llamada */
void usoLocal( void )
{
   int x = 25;  /* se inicializa cada vez que se llama usoLocal */

   printf( "\nla x local en usoLocal es %d despues de entrar a usoLocal\n", x );
   x++;
   printf( "la x local en usoLocal es %d antes de salir de usoLocal\n", x );
} /* fin de la función usoLocal */

/* usoStaticLocal inicializa la variable static local x sólo la primera vez
   que se invoca a la función; el valor de x se guarda entre las llamadas a esta
   función */
void usoStaticLocal( void )
{
   /* se inicializa sólo la primera vez que se invoca a usoStaticLocal */
   static int x = 50;

   printf( "\n la x local estatica es %d al entrar a usoStaticLocal\n", x );
   x++;
   printf( "la x local estatica  es %d al salir de usoStaticLocal\n", x );
} /* fin de la función usoStaticLocal */

/* la función usoGlobal modifica la variable global x durante cada llamada */
void usoGlobal( void )
{
   printf( "\nla x global es %d al entrar a usoGlobal\n", x );
   x *= 10;
   printf( "la x global es %d al salir de usoGlobal\n", x );
} /* fin de la función usoGlobal */
```

**Figura 5.12** Ejemplo de alcance. (Parte 2 de 3.)

```
la x local fuera del alcance de main es 5
la x local en el alcance interno de main es 7
la x local en el alcance externo de main es 5

la x local en usoLocal es 25 despues de entrar a usoLocal
la x local en usoLocal es 26 antes de salir de usoLocal

la x local estatica es 50 al entrar a usoStaticLocal
la x local estatica es 51 al salir de usoStaticLocal

la x global es 1 al entrar a usoGlobal
la x global es 10 al salir de usoGlobal

la x local en usoLocal es 25 despues de entrar a usoLocal
la x local en usoLocal es 26 antes de salir de usoLocal

la x local estatica es 51 al entrar a usoStaticLocal
la x local estatica x es 52 al salir de usoStaticLocal

la x global es 10 al entrar a usoGlobal
la x global es 100 al salir de usoGlobal

la x local en main es 5
```

**Figura 5.12** Ejemplo de alcance. (Parte 3 de 3.)

## 5.13 Recursividad

En general, los programas que ya explicamos están estructurados de tal modo que las funciones se llaman unas a otras de una manera disciplinada y jerárquica. Para algunos tipos de problemas, es útil tener funciones que se llaman a sí mismas. Una *función recursiva* es una función que se llama a sí misma de manera directa o indirecta a través de otra función. La recursividad es un tema complejo que se imparte en cursos de computación largos y avanzados. En esta sección y en la siguiente, explicaremos ejemplos sencillos sobre recursividad. El libro trata ampliamente la recursividad a lo largo de los capítulos 5 a 12. La figura 5.17 de la sección 5.15 resume los 31 ejemplos y ejercicios de recursividad contenidos en el libro.

Primero, consideraremos la recursividad de manera conceptual, y posteriormente explicaremos varios programas que contienen funciones recursivas. Los métodos para solucionar problemas por medio de la recursividad tienen algunos elementos en común. Se llama a una función recursiva para resolver un problema. La función en realidad sólo sabe cómo resolver el problema para el caso más sencillo, o *caso base*. Si se invoca a la función desde el caso base, ésta simplemente devuelve un resultado. Si se llama a la función desde un problema más complejo, la función divide el problema en dos partes conceptuales. Una parte que la función sabe cómo resolver y una parte que la función no sabe cómo resolver. Para hacer posible la recursividad, la segunda parte debe replantear el problema original, pero con una versión ligeramente más sencilla o más pequeña que el problema original. Debido a que este problema se parece al problema original, la función lanza (llama) a una nueva copia de sí misma para que trabaje con el problema más pequeño, a esto se le denomina *llamada recursiva* o también *paso recursivo*. El paso recursivo también incluye la palabra reservada **`return`**, debido a que su resultado se combinará con la parte del problema que la función sabe cómo resolver para formar un resultado que se pase a la llamada original a la función, posiblemente **`main`**.

El paso recursivo se ejecuta mientras la llamada a la función original permanezca abierta, es decir, mientras no termine su ejecución. El paso recursivo puede generar muchas más de estas llamadas recursivas, mientras la función continúa dividiendo cada problema en dos partes conceptuales. Para que la recursividad termine, cada vez que la función se invoca a sí misma con una versión del problema ligeramente más sencilla que el problema original, esta secuencia de problemas más pequeños debe converger en algún momento con el caso base. En ese punto, la función reconoce el caso base, devuelve el resultado a la copia previa de la función, y se

presenta una secuencia de resultados que se mueve hacia arriba, hasta que la función original devuelve el resultado final a **`main`**. Todo esto suena bastante extraño, si lo comparamos con el tipo de problemas en los que hemos utilizado las llamadas convencionales a las funciones utilizadas hasta este punto. De hecho, se necesita bastante práctica en la escritura de programas recursivos, antes de que el proceso logre obtener una apariencia natural. Para ejemplificar estos conceptos, escribamos un programa recursivo que realice un cálculo matemático muy popular.

El factorial de un entero no negativo *n*, se escribe *n*! (y se pronuncia "*n* factorial"), es el producto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

donde 1! es igual a 1, y 0! se define como 1. Por ejemplo, 5! Es el producto 5*4*3*2*1, el cual es igual a 120.

El factorial de un entero, **`numero`**,[1] mayor o igual que 0, se puede calcular de manera *iterativa* (no recursiva) por medio de una instrucción **`for`** de la siguiente manera:

```
factorial = 1;
for ( contador = numero; contador >= 1; contador-- )
    factorial *= contador;
```

Se puede llegar a una definición recursiva de la función factorial mediante la siguiente relación:

$$n! = n \cdot (n - 1)!$$

Por ejemplo, podemos ver claramente que 5! es lo mismo que 5*4!, como lo mostramos a continuación:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! se lleva a cabo como muestra la figura 5.13. La figura 5.13a muestra la manera en que proceden las llamadas recursivas hasta que 1! se evalúa como 1, lo cual termina la recursividad. La figura 5.13b muestra los valores devueltos por cada llamada recursiva a su llamada original, hasta que se calcula y se devuelve el valor final.

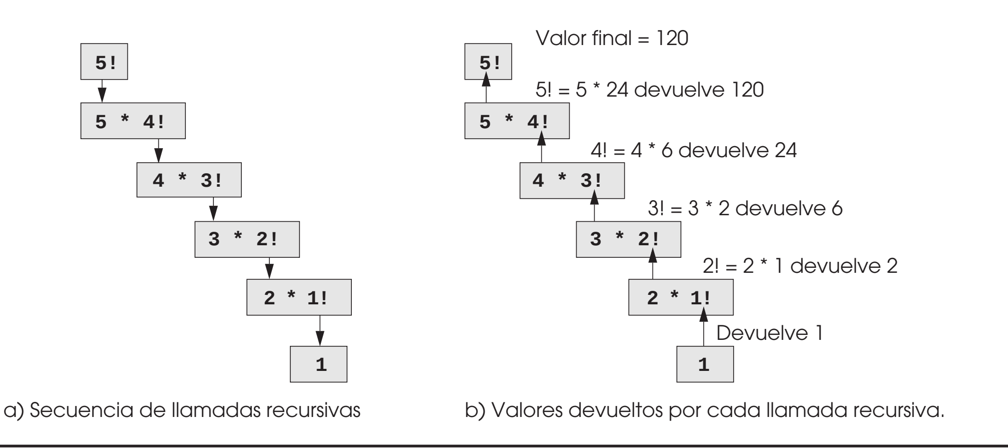



**Figura 5.13** Evaluación recursiva de 5!

La figura 5.14 utiliza la recursividad para calcular e imprimir los factoriales de los enteros de 0 a 10 (explicaremos por qué elegimos el tipo de dato **`long`**, más adelante). La función recursiva **`factorial`** primero evalúa si la condición de terminación es verdadera, es decir, si **`numero`** es menor o igual que 1. Si efectivamente, **`numero`** es menor o igual que 1, **`factorial`** devuelve 1; no se necesita mayor recursividad, y el programa termina. Si **`numero`** es mayor que 1, la instrucción

```
return numero * factorial(numero – 1);
```

1. Los acentos ortográficos no son permitidos en el compilador estándar de C. (Nota del revisor técnico.)

expresa el problema como el producto de `numero` y la llamada recursiva a la función `factorial` que evalúa el `factorial de numero – 1`. Observe que `factorial(numero – 1 )` es un problema ligeramente más sencillo que el cálculo original `factorial(numero)`.

La función `factorial` (línea 23) se declara para recibir un parámetro de tipo `long` y para devolver un resultado de tipo `long`. Ésta es una notación abreviada para `long int`. El estándar de C especifica que una variable de tipo `long int` se almacena en al menos 4 bytes, y por lo tanto puede almacenar un valor tan grande como +2147483647. Como podemos ver en la figura 5.14, lo valores factoriales crecen rápidamente. Elegimos el tipo de dato `long` de manera que el programa pueda calcular los factoriales mayores que 7! en computado-

```
1  /* Figura 5.14: fig05_14.c
2     Función factorial recursiva */
3  #include <stdio.h>
4
5  long factorial( long numero ); /* prototipo de la función */
6
7  /* la función main comienza la ejecución del programa */
8  int main()
9  {
10    int i; /* contador */
11
12    /* repite 11 veces; durante cada iteración, calcula
13       el factorial( i ) y despliega el resultado */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* fin de for */
17
18    return 0; /* indica terminación exitosa */
19
20 } /* fin de main */
21
22 /* definición recursiva de la función factorial */
23 long factorial( long numero )
24 {
25    /* caso base */
26    if ( numero <= 1 ) {
27       return 1;
28    } /* fin de if */
29    else { /* paso recursivo */
30       return ( numero * factorial( numero - 1 ) );
31    } /* fin de else */
32
33 } /* fin de la función factorial */
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

**Figura 5.14** Cálculo de factoriales con una función recursiva.

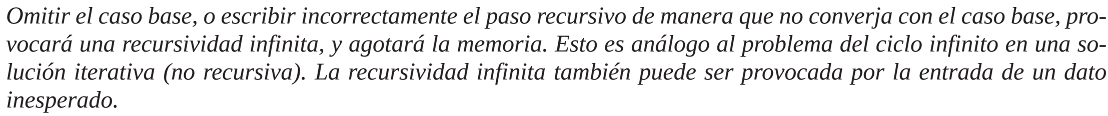
ras con enteros pequeños (por ejemplo de dos bytes). El especificador de conversión **`%ld`** se utiliza para imprimir valores de tipo **`long`**. Desafortunadamente, la función **`factorial`** crea valores grandes tan rápidamente, que incluso **`long int`** no nos ayuda a imprimir muchos valores factoriales antes de que excedamos el tamaño de la una variable **`long int`**.

Conforme analizamos los ejercicios, concluimos que podría ser necesario utilizar **`double`** para que el usuario pueda calcular números más grandes. Esto señala una debilidad de C (y muchos otros lenguajes de programación), a saber, que el lenguaje no está lo suficientemente extendido para manejar los requerimientos únicos de distintas aplicaciones. Como veremos más adelante, C++ es un lenguaje extensible que nos permite crear enteros arbitrariamente grandes, si así lo deseamos.

**Error común de programación 5.15**

*Olvidar devolver un valor desde una función recursiva cuando se necesita uno.*

**Error común de programación 5.16**



*Omitir el caso base, o escribir incorrectamente el paso recursivo de manera que no converja con el caso base, provocará una recursividad infinita, y agotará la memoria. Esto es análogo al problema del ciclo infinito en una solución iterativa (no recursiva). La recursividad infinita también puede ser provocada por la entrada de un dato inesperado.*

## 5.14 Ejemplo sobre cómo utilizar la recursividad: Serie de Fibonacci

La serie de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, …

comienza con 0 y 1, y tiene la propiedad de que cada número subsiguiente es la suma de los dos números anteriores de la serie.

La serie se presenta en la naturaleza y, en particular, describe la forma de una espiral. La razón de los números sucesivos de Fibonacci converge en un valor constante de 1.618…. Este número también se presenta repetidamente en la naturaleza y se le ha llamado la *razón dorada* o la *media dorada*. Los humanos tienden a describir a la media dorada como estéticamente agradable. Los arquitectos con frecuencia diseñan ventanas, habitaciones y edificios, cuya longitud y ancho se basan en la razón de la media dorada. Las tarjetas postales con frecuencia se diseñan con una razón de media dorada longitud/ancho.

La serie de Fibonacci puede definirse recursivamente de la siguiente manera:

*fibonacci(0) = 0*
*fibonacci(1) = 1*
*fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)*

La figura 5.15 calcula recursivamente el *enésimo* número de Fibonacci, utilizando la función **`fibonacci`**. Observe que los números de Fibonacci tienden a volverse grandes rápidamente. Por lo tanto, elegimos el tipo de dato **`long`** para el tipo de parámetro y para el tipo de retorno de la función **`fibonacci`**. En la figura 5.15, cada par de líneas de salida muestra una ejecución separada del programa.

```
/* Figura 5.15: fig05_15.c
   Función recursiva de fibonacci */
#include <stdio.h>

long fibonacci( long n ); /* prototipo de la función */

/* la función main comienza la ejecución del programa */
int main()
{
   long resultado;  /* valor fibonacci */
   long numero;     /* numero a introducir por el usuario */
```

**Figura 5.15** Generación recursiva de números de Fibonacci. (Parte 1 de 3.)

```

   /* obtiene un entero del usuario */
   printf( “Introduzca un entero: “ );
   scanf( “%ld”, &numero);

   /* calcula el valor fibonacci del número introducido por el usuario */
   resultado = fibonacci( numero );

   /* despliega el resultado */
   printf( “Fibonacci( %ld ) = %ld\n”, numero, resultado );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* definición de la función recursiva fibonacci */
long fibonacci( long n )
{
   /* caso base */
   if ( n == 0 || n == 1 ) {
      return n;
   } /* fin de if */
   else { /* paso recursivo */
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } /* fin de else */

} /* fin de la función fibonacci */
```

```
Introduzca un entero: 0
Fibonacci( 0 ) = 0
```

```
Introduzca un entero: 1
Fibonacci( 1 ) = 1
```

```
Introduzca un entero: 2
Fibonacci( 2 ) = 1
```

```
Introduzca un entero: 3
Fibonacci( 3 ) = 2
```

```
Introduzca un entero: 4
Fibonacci( 4 ) = 3
```

```
Introduzca un entero: 5
Fibonacci( 5 ) = 5
```

```
Introduzca un entero: 6
Fibonacci( 6 ) = 8
```

**Figura 5.15** Generación recursiva de números de Fibonacci. (Parte 2 de 3.)

```
Introduzca un entero: 10
Fibonacci( 10 ) = 55
```

```
Introduzca un entero: 20
Fibonacci( 20 ) = 6765
```

```
Introduzca un entero: 30
Fibonacci( 30 ) = 832040
```

```
Introduzca un entero: 35
Fibonacci( 35 ) = 9227465
```

**Figura 5.15** Generación recursiva de números de Fibonacci. (Parte 3 de 3.)

La llamada a **`fibonacci`** desde **`main`** no es una llamada recursiva (línea 18), pero todas las llamadas subsiguientes a **`fibonacci`** sí lo son (línea 35). Cada vez que se invoca a **`fibonacci`**, ésta inmediatamente evalúa el caso base; **`n`** es igual que **`0`** o **`1`**. Si esto es verdadero, **`n`** es devuelto. De manera interesante, si **`n`** es mayor que **`1`**, el paso de recursividad genera *dos* llamadas recursivas, cada una de las cuales es para un problema ligeramente más sencillo que la llamada original a **`fibonacci`**. La figura 5.16 muestra cómo es que la función **`fibonacci`** evaluaría a **`fibonacci(3)`**.

Esta cifra muestra algunas cuestiones interesantes sobre el orden en el que los compiladores de C evalúan los operandos de los operadores. Éste es un asunto diferente al del orden en el que los operadores se aplican a sus operandos, a saber, el orden dictado por las reglas de precedencia de los operadores. De la figura 5.16 se desprende que mientras se lleva a cabo la evaluación de **`fibonacci(3)`**, se harán dos llamadas recursivas, a saber, **`fibonacci(2)`** y **`fibonacci(1)`**. Pero, ¿en qué orden se harán estas llamadas? La mayoría de los programadores simplemente suponen que los operandos se evaluarán de izquierda a derecha. De manera extraña, el estándar de ANSI no especifica el orden en el que los operandos de los operadores (incluyendo +) se van a evaluar. Por lo tanto, el programador no debe hacer suposiciones con respecto al orden en que se ejecutarán estas llamadas. De hecho, las llamadas podrían ejecutar primero a **`fibonacci(2)`** y después a **`fibonacci(1)`**, o las llamadas podrían ejecutarse en el orden inverso, **`fibonacci(1)`** y después **`fibonacci(2)`**. En este programa, y en la mayoría de los programas, el resultado final será el mismo. Sin embargo, en algunos programas, la evaluación de un operando podría tener efectos colaterales que podrían afectar el resultado final de la expresión. Sobre los muchos operadores de C, el estándar de ANSI especifica el orden de evaluación de los operandos de sólo cuatro operadores, a saber, **&&**, **`||`**, el operador coma(**`,`**) y **`?:`**. Los tres primeros son operadores binarios, cuyos dos operandos

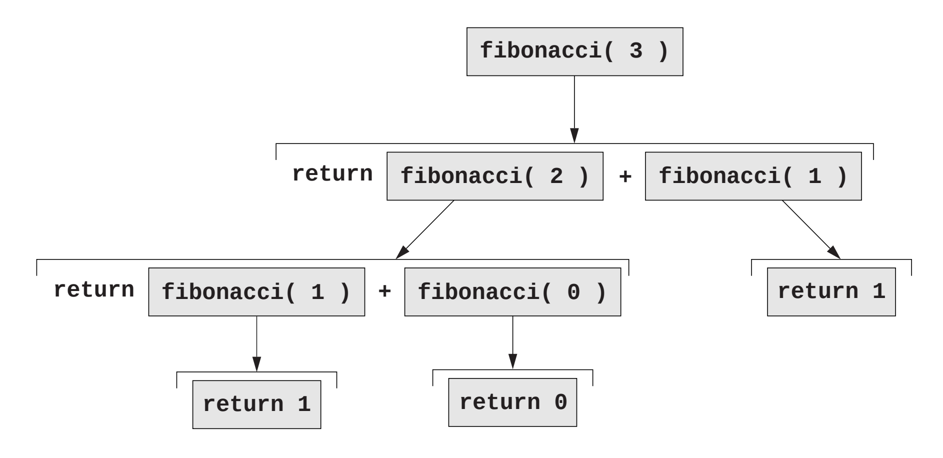



**Figura 5.16** Conjunto de llamadas recursivas a **`fibonacci( 3 )`**.

serán evaluados de izquierda a derecha con toda certeza. [*Nota:* Las comas utilizadas para separar los argumentos de una llamada a función, no son operadores coma.] El último operador es el único operador ternario de C. El operando que se encuentra más a su izquierda siempre se evalúa primero; si dicho operando arroja un valor diferente de cero, el operando de en medio se evalúa después, y el último operando se ignora; si el operando que se encuentra más a la izquierda del operado arroja un valor cero, el tercer operando se evalúa después y el operando de en medio se ignora.



**Error común de programación 5.17**

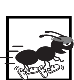
*Escribir programas que dependan del orden de evaluación de operandos correspondientes a operadores diferentes a* `&&`, `||`, `?:`, *y el operador coma* (`,`)*, pueden ocasionar errores, ya que los compiladores podrían no evaluar los operandos en el orden que el programador espera.*



**Tip de portabilidad 5.2**

*Los programas que dependen del orden de evaluación de operandos correspondientes a operadores diferentes a* `&&`, `||`, `?:`, *y el operador coma* (`,`)*, pueden funcionar de manera diferente en sistemas con distintos compiladores.*

Es necesario que tenga cuidado cuando trabaje con programas recursivos como el que utilizamos aquí para generar números de Fibonacci. Cada nivel de recursividad de la función `fibonacci` tiene un efecto duplicativo sobre el número de llamadas, es decir, el número de llamadas recursivas que se ejecutarán para calcular en *enésimo* número de Fibonacci es del orden de $2^n$. Esto rápidamente se sale de control. Tan solo calcular el $20^{avo}$ número de Fibonacci requeriría alrededor de $2^{20}$ o un millón de llamadas, calcular el $30^{avo}$ número de Fibonacci requeriría alrededor de $2^{30}$ o mil millones de llamadas, y así sucesivamente. Los científicos de la computación se refieren a esto como *complejidad exponencial*. ¡Los problemas de esta naturaleza hacen temblar incluso a las computadoras más poderosas del mundo! Cuestiones de complejidad en general, y la complejidad exponencial en particular, se explican con detalle en cursos de ciencias de la computación de nivel avanzado, generalmente llamados "Algoritmos".



**Tip de rendimiento 5.4**

*Evite programas recursivos del estilo de Fibonacci, que resultan en una "explosión" exponencial de llamadas.*

## 5.15 Recursividad *versus* iteración

En las secciones anteriores, estudiamos dos funciones que pueden implementarse fácilmente, ya sea recursiva o iterativamente. En esta sección comparamos los dos métodos y explicamos por qué el programador podría elegir un método sobre el otro, en una situación particular.

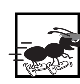
Tanto la iteración como la recursividad se basan en una estructura de control: la iteración utiliza una estructura de repetición; la recursividad utiliza una estructura de selección. Tanto la iteración como la recursividad involucran la repetición: la iteración utiliza explícitamente una estructura de repetición; la recursividad consigue la repetición a través de llamadas repetidas a función. Tanto la iteración como la recursividad involucran una prueba de terminación: la iteración termina cuando la condición de continuación de ciclo falla; la recursividad termina cuando se reconoce un caso base. La iteración con repetición controlada por contador y la recursividad gradualmente alcanzan la terminación: la iteración continúa modificando al contador hasta que éste toma el valor que hace que la condición de continuación de ciclo falle; la recursividad continúa produciendo versiones más sencillas del problema original, hasta que se logra el caso base. Tanto la iteración como la recursividad pueden durar infinitamente: en la iteración ocurre un ciclo infinito si la prueba de continuación de ciclo nunca se vuelve falsa; la recursividad infinita ocurre si el paso de recursividad no reduce el problema cada vez, de manera que converja en el caso base.

La recursividad tiene sus inconvenientes. Invoca de manera repetida al mecanismo, y por consecuencia, la sobrecarga de llamadas a la función. Esto puede ser costoso tanto en tiempo de proceso como en espacio de memoria. Cada llamada recursiva crea una copia de la función (en realidad sólo las variables de la función); esto puede consumir una considerable cantidad de memoria. Por lo general, la iteración ocurre dentro de una función, de manera que se evita la sobrecarga de llamadas repetidas a una función y la asignación adicional de memoria. Entonces, ¿por qué elegir la recursividad?



### Observación de ingeniería de software 5.13

*Cualquier problema que se pueda resolver de manera recursiva también se puede resolver de manera iterativa (no recursiva). Por lo general, el método de recursividad se elige por encima de uno de iteración, cuando el método de recursividad refleja de manera más natural el problema y genera un programa que es más fácil de entender y depurar. Otra razón para elegir una solución recursiva es que la solución iterativa no es obvia.*



### Tip de rendimiento 5.5

*Evite reutilizar la recursividad en situaciones de rendimiento. Las llamadas recursivas toman tiempo y consumen espacio adicional de memoria.*



### Error común de programación 5.18

*Tener una función no recursiva que de manera accidental se llama a sí misma directa o indirectamente a través de otra función.*

La mayoría de lo libros de programación introducen la recursividad mucho más adelante de lo que nosotros lo hacemos. Sentimos que la recursividad es un tema lo suficientemente rico y complejo como para presentarlo lo antes posible y distribuir ejemplos en el resto del libro. La figura 5.17 resume, por capítulo, los 31 ejemplos sobre recursividad y los ejercicios en el libro.

Cerremos este capítulo con algunas observaciones que hicimos repetidamente a lo largo del libro. La buena ingeniería de software es importante. El alto rendimiento es importante. Desafortunadamente, a menudo estas metas se contradicen una a otra. La buena ingeniería de software es la clave para hacer más manejable la tarea de desarrollar los sistemas de cómputo más grandes y complejos que necesitamos. El alto rendimiento es la clave para realizar los sistemas del futuro que demandarán cada vez más del hardware. ¿En dónde encajan las funciones aquí?

| Capítulo | Ejemplos de recursividad y ejercicios |
|---|---|
| *Capítulo 5* | Función factorial |
| | Función fibonacci |
| | Máximo común divisor |
| | Suma de dos enteros |
| | Multiplicación de dos enteros |
| | Cómo elevar un entero a una potencia entera |
| | Las torres de Hanoi |
| | `main` recursivo |
| | Impresión inversa de entradas desde el teclado |
| | Cómo visualizar la recursividad |
| *Capítulo 6* | Suma de los elementos de un arreglo |
| | Impresión de un arreglo |
| | Impresión inversa de un arreglo |
| | Impresión inversa de una cadena |
| | Verificar si una cadena es un palíndromo |
| | Valor mínimo de un arreglo |
| | Ordenamiento por selección |
| | Quicksort |
| | Búsqueda lineal |
| | Búsqueda binaria |
| *Capítulo 7* | Ocho reinas |
| | Recorrido de laberintos |

**Figura 5.17** Ejemplos y ejercicios de recursividad en el libro. (Parte 1 de 2.)

| Capítulo | Ejemplos de recursividad y ejercicios |
|---|---|
| *Capítulo 8* | Impresión inversa de una cadena introducida desde el teclado |
| *Capítulo 12* | Inserción en una lista ligada |
| | Eliminación en una lista ligada |
| | Búsqueda en una lista ligada |
| | Impresión inversa de una lista ligada |
| | Inserción en un árbol binario |
| | Recorrido en preorden de un árbol binario |
| | Recorrido en inorden de un árbol binario |
| | Recorrido postorden de un árbol binario |

**Figura 5.17** Ejemplos y ejercicios de recursividad en el libro. (Parte 2 de 2.)



**Tip de rendimiento 5.6**

*Funcionalizar los programas de manera sencilla y jerárquica, promueve la buena ingeniería de software. Sin embargo, tiene un precio. Un programa altamente funcionalizado, comparado con un programa monolítico (es decir, de una sola pieza) sin funciones, hace un gran número de llamadas a funciones y esto consume tiempo de ejecución en el procesador de la computadora. Sin embargo, aunque los programas monolíticos se ejecutan mejor, son más difíciles de programar, probar, corregir, mantener y evolucionar.*

Por lo tanto, funcionalice sus programas de manera juiciosa, y siempre tenga en mente el delicado balance entre el rendimiento y la buena ingeniería de software.

## RESUMEN

- La mejor manera de desarrollar y dar mantenimiento a un programa grande es dividiéndolo en varios módulos de programa más pequeños, cada uno de los cuales es más manejable que el programa original. En C, los módulos se escriben como funciones.
- Una función se invoca mediante una llamada a dicha función. La llamada a la función menciona a la función por su nombre y proporciona información (como argumentos) que la función necesita para realizar su tarea.
- El propósito del ocultamiento de información es que las funciones tengan acceso sólo a la información que necesitan para completar sus tareas. Ésta es una manera de implementar el principio del menor privilegio; uno de los principios más importantes de la buena ingeniería de software.
- Por lo general, las funciones se invocan en un programa, escribiendo el nombre de la función seguido por un paréntesis izquierdo, seguido por el argumento de la función (o una lista de argumentos separada por comas) y por el paréntesis derecho.
- El tipo de dato **`double`** es un tipo de dato de punto flotante como **`float`**. Una variable de tipo **`double`** puede almacenar un valor mucho más grande en magnitud y precisión que un número de tipo **`float`**.
- Cada argumento de función puede ser una constante, una variable, o una expresión.
- A una variable local se le conoce sólo en la definición de la función. Las otras funciones no están autorizadas para conocer el nombre de las variables locales de dichas funciones, y tampoco están autorizadas para conocer los detalles de implementación de cualquier otra función.
- El formato general para la definición de una función es:

  *tipo-valor-retorno nombre-función*( *lista-parámetros* )<br>
  {<br>
  *cuerpo-función*<br>
  }

- El *tipo-valor-retorno* especifica el tipo de valor devuelto por la función a la que se invoca. Si una función no devuelve valor, el *tipo-valor-retorno* se declara como **`void`**. El *nombre-función* es cualquier identificador válido. La *lista-parámetros* es una lista separada por comas que contiene las definiciones de las variables que se pasarán a la función. Si una fun-

ción no recibe valor alguno, *lista-parámetros* se declara como **void**. El c*uerpo-función* es un conjunto de definiciones e instrucciones que constituyen la función.

- Los argumentos que se pasan a una función deben coincidir en número, tipo y orden con los parámetros en la definición de la función.
- Cuando un programa encuentra una llamada a una función, el control se transfiere desde el punto de invocación hasta la función que fue invocada, las instrucciones de la función invocada se ejecutan y el control regresa a la invocación de la función.
- Una función invocada puede devolver el control al punto de invocación de tres maneras. Si la función no devuelve valor alguno, el control se devuelve cuando ésta alcanza la llave derecha que indica el fin de la función, o por medio de la ejecución de la instrucción:

```
return;
```

Si la función devuelve un valor, la instrucción:

**return** *expresión***;**

devuelve el valor de la *expresión*.

- El prototipo de una función declara el tipo de retorno de la función y declara el número, los tipos, y el orden de los parámetros que la función espera recibir.
- Los prototipos de las funciones permiten al compilador verificar que las funciones se invocan de manera correcta.
- El compilador ignora los nombres de variables mencionadas en el prototipo de la función.
- Cada biblioteca estándar tiene un encabezado correspondiente, el cual contiene los prototipos de todas las funciones de la biblioteca, así como las definiciones de las distintas constantes simbólicas necesarias para dichas funciones.
- Los programadores pueden crear e incluir sus propios encabezados.
- Cuando un argumento se pasa por valor, se crea una copia del valor de la variable y dicha copia se pasa a la función invocada. Los cambios que se hagan a esta copia dentro de la función no afectan el valor de la variable original.
- En C, todas las llamadas se hacen por valor.
- La función **rand** genera un entero entre **0** y **RAND_MAX**, el cual se define en C para ser al menos 32767.
- Los prototipos de las funciones **rand** y **srand** se encuentran en **<stdlib.h>**.
- Los valores producidos por **rand** se pueden escalar y modificar para crear valores dentro de un rango específico.
- Para randomizar un programa, utilice la función **srand** de la biblioteca estándar de C.
- Por lo general, la llamada a la función **srand** se inserta en un programa, una vez que éste se depuró totalmente. Durante la depuración, es mejor omitir **srand**. Esto garantiza la repetición de valores, lo cual es esencial para asegurarse de que las correcciones al programa de generación de números aleatorios funcionan adecuadamente.
- Para crear números aleatorios sin tener que introducir una semilla cada vez, utilizamos **srand(time(NULL))**. La función **time** devuelve el número de segundos que han pasado desde que inició el día. El prototipo de la función se localiza en el encabezado **<time.h>**.
- La ecuación general para escalar y modificar un número aleatorio es:

```
n = a + rand() % b;
```

donde **a** es el valor de cambio (es decir, el primer número del rango deseado de enteros consecutivos), y **b** es el factor de escalamiento (es decir, la longitud del rango de los enteros consecutivos).

- Una enumeración, introducida mediante la palabra reservada **enum**, es un conjunto de enteros constantes representado mediante identificadores. Los valores en un **enum** comienzan con **0** y se incrementan en **1**. También es posible asignar un valor entero a cada identificador en un **enum**. Los identificadores de una enumeración deben ser únicos, pero los valores pueden ser duplicados.
- En un programa, cada identificador tiene los atributos clase de almacenamiento, duración del almacenamiento, alcance y vinculación.
- C proporciona cuatro clases de almacenamiento indicadas mediante los especificadores de clase de almacenamiento: **auto**, **register**, **extern** y **static**.
- La duración de almacenamiento de un identificador se refiere al tiempo de existencia de un identificador en memoria.
- El alcance de un identificador se refiere al lugar en el que se puede hacer referencia a un identificador dentro del programa.
- La vinculación de un identificador determina, para un programa con múltiples archivos fuente, si un identificador es reconocido sólo en el archivo fuente actual o en cualquier archivo fuente mediante las declaraciones apropiadas.

- Las variables con duración automática de almacenamiento se crean cuando se entra al bloque en el que fueron definidas; éstas existen mientras el bloque se encuentra activo y se destruyen cuando se abandona el bloque. Por lo general, las variables locales de una función tienen una duración automática de almacenamiento.
- El especificador de clase de almacenamiento **`register`** puede colocarse antes de la declaración de una variable automática para sugerir al compilador que mantenga la variable en uno de los registros de alta velocidad del hardware de la computadora. El compilador podría ignorar las declaraciones **`register`**. La palabra reservada **`register`** solamente se puede utilizar con variables de duración automática de almacenamiento.
- Las palabras reservadas **`extern`** y **`static`** se utilizan para declarar identificadores para las variables y las funciones de duración estática de almacenamiento.
- Las variables con duración estática de almacenamiento se asignan y se inicializan una vez que comienza la ejecución del programa.
- Existen dos tipos de identificadores con duración estática de almacenamiento: identificadores externos (tales como variables globales y nombres de función) y variables locales declaradas con el identificador de clase de almacenamiento **`static`**.
- Las variables globales se crean al colocar las definiciones fuera de cualquier definición de función. Las variables globales retienen sus valores a través de la ejecución del programa.
- Las variables locales declaradas como **`static`** retienen su valor a lo largo de las llamadas a la función en la que se definieron.
- Todas las variables numéricas de duración estática de almacenamiento se inicializan en cero si el programador no las inicializa explícitamente.
- Los cuatro tipos de alcance para un identificador son: alcance de función, alcance de archivo, alcance de bloque y alcance de prototipo de función.
- Las etiquetas son los únicos identificadores con alcance de función. Las etiquetas se pueden utilizar en cualquier parte de la función en la que aparecen, pero no se puede hacer referencia a ellas fuera del cuerpo de la función.
- Un identificador declarado fuera de cualquier función tiene alcance de archivo. Dicho identificador es “conocido” en todas las funciones desde el punto en el que se declara el identificador y hasta el punto en el que termina el archivo.
- Los identificadores que se definen dentro de un bloque tienen alcance de bloque. El alcance de bloque termina al alcanzar la llave derecha de terminación de bloque (}).
- Las variables definidas al principio de una función tienen alcance de bloque, así como los parámetros de ésta, los cuales son considerados como variables locales por la función.
- Cualquier bloque puede contener definiciones de variables. Cuando los bloques están anidados, y un identificador en el bloque externo tiene el mismo nombre que un identificador en el bloque interno, el identificador en el bloque externo se mantiene “oculto” hasta que el bloque interno termina.
- Los únicos identificadores con alcance de prototipo de función son los que se utilizan en la lista de parámetros de un prototipo de función. Los identificadores utilizados en el prototipo de una función pueden reutilizarse en cualquier parte del programa sin crear ambigüedades.
- Una función recursiva es una función que se invoca a sí misma de manera directa o indirecta.
- Si una función recursiva se invoca mediante un caso base, la función simplemente devuelve un resultado. Si la función se invoca a sí misma mediante un problema más complejo, la función divide el problema en dos partes conceptuales. Una pieza que la función sabe cómo resolver y una versión más sencilla del problema original. Debido a que este nuevo problema es similar al problema original, la función lanza una llamada recursiva para trabajar con el problema más sencillo.
- Para que la recursividad termine, cada vez que la función recursiva se invoca a sí misma por medio de un problema más sencillo que el problema original, la secuencia de problemas cada vez más sencillos debe converger en el caso base. Cuando la función reconoce el caso base, devuelve el resultado a la función llamada previamente, y se origina una secuencia de resultados hacia arriba hasta que la llamada a la función original devuelve el resultado final.
- El estándar de ANSI no especifica el orden en el que se evalúan los operandos de la mayoría de los operadores (incluso +). De los muchos operadores de C, el estándar especifica el orden de evaluación de los operandos de los operadores **`&&`**, **`||`**, el operador coma (**`,`**) y **`?:`**. Los primeros tres son operadores binarios cuyos dos operandos se evalúan de izquierda a derecha. El último operador es el único operador ternario de C. Su operando más a la izquierda se evalúa primero; si dicho operando da como resultado un número diferente de cero, el operando de en medio se evalúa a continuación y el último operando se ignora; si el operando más a la izquierda da como resultado cero, a continuación se evalúa el tercer operando y el operador que se encuentra en el centro se ignora.
- Tanto la iteración como la recursividad se basan en una estructura de control: la iteración utiliza una estructura de repetición; y la recursividad utiliza una estructura de selección.

- Tanto la iteración como la recursividad involucran la repetición: la iteración utiliza de manera explícita una estructura de repetición; la recursividad logra la repetición a través de llamadas repetidas a una función.
- La iteración y la recursividad involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursividad termina cuando se reconoce el caso base.
- La iteración y la recursividad pueden repetirse indefinidamente: en el caso de la iteración ocurre un ciclo infinito si la condición de continuación de ciclo nunca se hace falsa; la recursividad infinita ocurre si el paso recursivo no reduce el problema de manera que converja en el caso base.
- La recursividad invoca de manera repetida al mecanismo, y por consecuencia se presenta una sobrecarga de llamadas a función. Esto puede costar caro en cuanto a tiempo de proceso y a espacio de memoria.

## TERMINOLOGÍA

abstracción
alcance
alcance de archivo
alcance de bloque
alcance de función
alcance de prototipo de función
almacenamiento automático
argumento en una llamada a función
biblioteca estándar de C
bloque
caso base en la recursividad
clases de almacenamiento
coerción de argumentos
compilador optimizado
copia de un valor
definición de una función
divide y vencerás
duración automática de almacenamiento
duración de almacenamiento
efectos colaterales
encabezado
encabezados de la biblioteca estándar
**enum** (enumeración)
escalamiento
especificador de clase de almacenamiento
especificador de clase de almacenamiento **auto**
especificador de clase de almacenamiento **extern**
especificador de clase de almacenamiento **register**
especificador de clase de almacenamiento **static**
especificador de conversión **%s**
expresiones mixtas
función
función definida por el programador
función factorial
función invocada
función que llama
función recursiva
funciones matemáticas de la biblioteca
generación de números aleatorios
ingeniería de software
invocar a una función
iteración
jerarquía de promoción
lista de parámetros
llamada a una función
llamada por referencia
llamada por valor
llamada recursiva
llamar a una función
modificación
números pseudoaleatorios
ocultamiento de información
principio del menor privilegio
prototipo de función
**rand**
**RAND_MAX**
randomizar
recursividad
**return**
simulación
**srand**
**time**
tipo del valor de retorno
**unsigned**
variable automática
variable global
variable local
variable **static**
vinculación
**void**

## ERRORES COMUNES DE PROGRAMACIÓN

**5.1** Omitir tipo-valor-retorno en una definición de función es un error de sintaxis si el prototipo de la función especifica un tipo diferente a **int**.

**5.2** Olvidar devolver un valor desde la función cuando se supone que se debe retornar alguno, puede provocar errores inesperados. El C estándar establece que el resultado de esta omisión es indefinido.

**5.3** Devolver un valor desde una función, con un tipo de retorno **void**, es un error de sintaxis.

**5.4** Especificar los parámetros de la función del mismo tipo como **double x**, **y**, en lugar de hacerlo como **double x**, **double y**, podría provocar errores en sus programas. La declaración de parámetros como **double x**, **y**, en realidad hará que **y** sea un parámetro de tipo **int**, ya que **int** es el tipo predeterminado.

**5.5** Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de la definición de una función, es un error de sintaxis.

**5.6** Definir otra vez un parámetro de función como una variable local dentro de la función, es un error de sintaxis.

**5.7** Definir una función dentro de otra función es un error de sintaxis.

**5.8** Olvidar el punto y coma al final del prototipo de la función, es un error de sintaxis.

**5.9** Convertir un tipo de dato de mayor nivel en la jerarquía a uno de menor nivel, puede modificar el valor del dato.

**5.10** Olvidar el prototipo de una función provoca un error de sintaxis si en el programa el tipo del valor de retorno no es **`int`** y la definición de la función aparece después de la llamada a la función. De lo contrario, olvidar un prototipo de función puede provocar errores en tiempo de ejecución y un resultado inesperado.

**5.11** Usar **`srand`** en un lugar de **`rand`** para generar números aleatorios.

**5.12** Asignar un valor a una constante de enumeración después de que se define es un error de sintaxis.

**5.13** Utilizar múltiples especificadores de clase de almacenamiento para un identificador. Sólo se puede aplicar un especificador de clase de almacenamiento a un identificador.

**5.14** Utilizar de manera accidental el mismo nombre para un identificador en un bloque interno y en un bloque externo, cuando de hecho, el programador quiere que el identificador del bloque externo se encuentre activo durante la ejecución del bloque interno.

**5.15** Olvidar devolver un valor desde una función recursiva cuando se necesita uno.

**5.16** Omitir el caso base, o escribir incorrectamente el paso recursivo de manera que no converja con el caso base, provocará una recursividad infinita, y agotará la memoria. Esto es análogo al problema del ciclo infinito en una solución iterativa (no recursiva). La recursividad infinita también puede ser provocada por la entrada de un dato inesperado.

**5.17** Escribir programas que dependan del orden de evaluación de operandos correspondientes a operadores diferentes a **`&&`**, **`||`**, **`?:`**, y el operador coma (**`,`**), pueden ocasionar errores, ya que los compiladores podrían no evaluar los operandos en el orden que el programador espera.

**5.18** Tener una función no recursiva que de manera accidental se llama a sí misma directa o indirectamente a través de otra función.

## *TIPS PARA PREVENIR ERRORES*

**5.1** Cuando utilice las funciones matemáticas de la biblioteca, incluya el encabezado math por medio de la directiva de preprocesador **`#include <math.h>`**.

**5.2** Evite nombres de variables que oculten nombres con alcances externos. Esto se puede llevar a cabo simplemente evitando el uso de identificadores duplicados en un programa.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**5.1** Conozca la rica colección de funciones de la biblioteca estándar de C.

**5.2** Coloque una línea en blanco entre las definiciones de las funciones para separarlas y mejorar la legibilidad del programa.

**5.3** Aun cuando un tipo de retorno omitido devuelve de manera predeterminada un **`int`**, siempre establezca el tipo de retorno de manera explícita.

**5.4** Incluya el tipo de cada parámetro en la lista de parámetros, incluso si el parámetro es del tipo predeterminado **`int`**.

**5.5** Aunque no es incorrecto hacerlo, en la definición de la función no utilice el mismo nombre para los argumentos que se pasan a una función y para sus parámetros correspondientes. Esto ayuda a evitar la ambigüedad.

**5.6** Elegir nombres significativos de funciones y de parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.

**5.7** Incluya los prototipos de todas las funciones, para aprovechar las capacidades de verificación de tipos de C. Utilice la directiva de preprocesador **`#include`** para obtener los prototipos de función correspondientes a las funciones de la biblioteca estándar, a partir de los encabezados en las bibliotecas apropiadas, o para obtener encabezados que contengan prototipos de funciones desarrolladas por usted y/o sus compañeros de grupo.

**5.8** En ocasiones, para efectos de documentación, los nombres de parámetros se incluyen en los prototipos de las funciones (así lo preferimos nosotros). El compilador ignora estos nombres.

**5.9** Utilice sólo letras mayúsculas en los nombres de las constantes de enumeración para hacer que resalten en el programa, y para indicar que las constantes de enumeración no son variables.

## *TIPS DE RENDIMIENTO*

**5.1** El almacenamiento automático es una manera de ahorrar memoria, ya que las variables automáticas existen sólo cuando son requeridas. Éstas se crean cuando la función en la que se definen inicia su ejecución, y se destruyen cuando termina su ejecución.

**5.2** El especificador **`register`** puede colocarse antes de la declaración de la variable automática para sugerir al compilador que mantenga a la variable en uno de los registros de hardware de alta velocidad. Si utiliza de manera intensa variables tales como contadores o totales, éstas pueden mantenerse en registros de hardware; de este modo podrá evitar la sobrecarga producida por pasar las variables de la memoria a los registros y almacenar nuevamente los registros en memoria.

**5.3** A menudo, las declaraciones **`register`** son innecesarias. Con frecuencia, los compiladores optimizados actuales son capaces de reconocer las variables utilizadas y pueden decidir colocarlas en registros, sin necesidad de que el programador las declare como **`register`**.

**5.4** Evite programas recursivos del estilo de Fibonacci, que resultan en una "explosión" exponencial de llamadas.

**5.5** Evite utilizar la recursividad en situaciones de rendimiento. Las llamadas recursivas toman tiempo y consumen espacio adicional de memoria.

**5.6** Funcionalizar los programas de manera sencilla y jerárquica promueve la buena ingeniería de software. Sin embargo, tiene un precio. Un programa altamente funcionalizado, comparado con un programa monolítico (es decir, en una sola pieza) sin funciones, hace un gran número de llamadas a funciones y esto consume tiempo de ejecución en el procesador de la computadora. Sin embargo, aunque los programas monolíticos se ejecutan mejor, son más difíciles de programar, probar, corregir, mantener y evolucionar.

## *TIPS DE PORTABILIDAD*

**5.1** Utilizar funciones de la biblioteca estándar de C hace que los programas sean más portables.

**5.2** Los programas que dependen del orden de evaluación de operandos correspondientes a operadores diferentes a **`&&`**, **`||`**, **`?:`**, y el operador coma (**`,`**), pueden funcionar de manera diferente en sistemas con distintos compiladores.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**5.1** Evite "reinventar la rueda". Cuando sea posible, utilice las funciones de la biblioteca estándar de C en lugar de escribir nuevas funciones. Esto puede reducir el tiempo de desarrollo de un programa.

**5.2** En los programas que contienen muchas funciones, a menudo **`main`** se implementa como un grupo de llamadas a funciones que realizan el grueso del trabajo del programa.

**5.3** Cada función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar de manera clara dicha tarea. Esto facilita la abstracción y promueve la reutilización de software.

**5.4** Si usted no puede elegir un nombre conciso que exprese lo que hace la función, es posible que su función intente realizar demasiadas tareas. Por lo general, es mejor dividir dicha función en varias funciones más pequeñas.

**5.5** Una función no debe ser más grande que una página. Mejor aún, una función no debe ser más grande que la mitad de una página. Las funciones pequeñas promueven la reutilización de software.

**5.6** Los programas deben escribirse como colecciones de funciones pequeñas. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.

**5.7** Una función que tiene un gran número de parámetros podría realizar demasiadas tareas. Considere el dividirla en funciones más pequeñas para realizar tareas separadas. El encabezado de la función debe caber, si es posible, en una sola línea.

**5.8** El prototipo de una función, el encabezado de la función y las llamadas a la función deben concordar en número, tipo, orden de argumentos y parámetros, y en el tipo del valor de retorno.

**5.9** Un prototipo de función que se coloca fuera de la definición de cualquier función se aplica a todas las llamadas a la función que aparecen después del prototipo de función en el archivo. Un prototipo de función que se coloca en la función se aplica sólo a las llamadas que se hacen en dicha función.

**5.10** El almacenamiento automático es un ejemplo del principio del menor privilegio, permite el acceso a los datos sólo cuando es absolutamente necesario. ¿Para que tener variables almacenadas en memoria y accesibles si, de hecho, no son necesarias?

**5.11** Definir una variable como global, en lugar de hacerlo como local, permite que ocurran efectos colaterales, por ejemplo, cuando una función que no necesita acceso a la variable la modifica de manera accidental o maliciosa. En general, debe evitarse el uso de variables globales, excepto en ciertas situaciones con requerimientos especiales de rendimiento (como explicaremos en el capítulo 14).

**5.12** Las variables que se utilizan sólo en una función en particular, deben definirse como variables locales en esa función y no como variables externas.

**5.13** Cualquier problema que se pueda resolver de manera recursiva también se puede resolver de manera iterativa (no recursiva). Por lo general, el método de recursividad se elige por encima de uno de iteración, cuando el método de recursividad refleja de manera más natural el problema y genera un programa que es más fácil de entender y depurar. Otra razón para elegir una solución recursiva es que la solución iterativa no es obvia.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**5.1** Responda cada una de las siguientes preguntas:

a) A un módulo de programa en C, se le llama __________.
b) Una función se invoca mediante una __________.
c) A una variable que sólo se conoce dentro de la función en la que se definió se le llama __________.
d) La instrucción __________ dentro de una función se utiliza para pasar el valor de una expresión hacia la función que la invoca.
e) La palabra reservada __________ se utiliza dentro de una función para indicar que ésta no devuelve valor alguno, o para indicar que la función no contiene parámetros.
f) El __________ de un identificador se refiere a la porción del programa en la que se puede utilizar dicho identificador.
g) Las tres formas de devolver el control desde la función invocada hasta la función que llama son __________, __________ y __________.
h) Un __________ permite al compilador verificar el número, tipo y orden de los argumentos que se pasan a una función.
i) La función __________ se utiliza para producir números aleatorios.
j) La función __________ se utiliza para establecer la semilla de los números aleatorios para randomizar un programa.
k) Los especificadores de clase de almacenamiento son: __________, __________, __________ y __________.
l) Se asume que las variables declaradas dentro de un bloque, o en la lista de parámetros de una función, tienen una clase de almacenamiento __________, a menos que se especifique lo contrario.
m) El especificador de clase de almacenamiento __________ es una recomendación al compilador para que almacene una variable en uno de los registros de la computadora.
n) Una variable definida fuera de cualquier bloque o función es una variable __________.
o) Para que una variable local de una función retenga su valor entre las llamadas a la misma, la variable se debe declarar con el especificador de clase de almacenamiento __________.
p) Los cuatro posibles alcances de un dentificador son __________, __________, __________ y __________.
q) Una función que se invoca a sí misma de manera directa o indirecta es una función __________.
r) Por lo general, una función recursiva tiene dos componentes: uno que proporciona un medio para que termine la recursividad a través de la evaluación de un caso __________, y otro que expresa el problema como una llamada recursiva a un problema ligeramente más sencillo que el de la llamada original.

**5.2** Para el siguiente programa, establezca el alcance (si es alcance de función, de archivo, de bloque o de prototipo de función) de cada uno de los siguientes elementos.

a) La variable **`x`** en **`main`**.
b) La variable **`y`** en **`cubo`**.
c) La función **`cubo`**.
d) La función **`main`**.
e) El prototipo de la función para **`cubo`**.
f) El identificador y en el prototipo de la función **`cubo`**.

```
#include <stdio.h>
int cubo( int y );

int main( )
{
   int x;

   for ( x = 1; x <= 10; x++ )
```

(*continúa en la siguiente página*)

```
        printf( "%d\n", cubo( x ) );
    return 0;
}

int cubo( int y )
{
    return y * y * y;
}
```

**5.3** Escriba un programa que compruebe si los ejemplos sobre las llamadas a las funciones matemáticas de la biblioteca que mostramos en la figura 5.2 producen realmente los resultados indicados.

**5.4** Indique el encabezado para cada una de las siguientes funciones.

a) La función **hipotenusa** que toma dos argumentos de punto flotante de precisión doble, **lado1** y **lado2**, y devuelve un resultado de punto flotante de precisión doble.
b) La función **elMenor** que toma tres enteros, **x**, **y**, **z**, y devuelve un entero.
c) La función **instrucciones** que no recibe argumentos y no devuelve valor alguno. [*Nota:* Por lo general, dichas funciones se utilizan para desplegar instrucciones para el usuario.]
d) La función **intAfloat** que toma un argumento entero, **numero**, y devuelve un resultado en punto flotante.

**5.5** Escriba el prototipo para cada una de las siguientes:

a) La función descrita en el ejercicio 5.4a.
b) La función descrita en el ejercicio 5.4b.
c) La función descrita en el ejercicio 5.4c.
d) La función descrita en el ejercicio 5.4d.

**5.6** Escriba una declaración para lo siguiente:

a) El entero **cuenta** que debe mantenerse en un registro. Inicialice **cuenta** con **0**.
b) La variable de punto flotante **ultValor** que debe retener su valor entre las llamadas a la función en la que se definió.
c) El entero externo **numero** cuyo alcance debe restringirse al resto del archivo en el que se definió.

**5.7** Encuentre el error en cada uno de los segmentos de programa y explique cómo puede corregir dicho error (vea también el ejercicio 5.50):

a)
```
int g( void )
{
  printf( "Dentro de la funcion g\n" );
  int h( void )
  {
    printf( "Dentro de la función h\n" );
  }
}
```
b)
```
int suma( int x, int y )
{
  int resultado;
  resultado = x + y;
}
```
c)
```
int suma( int n )
{
  if ( n == 0 )
    return 0;
  else
    n + suma( n – 1 );
}
```
d)
```
void f( float a );
{
  float a;
  printf( "%f", a );
}
```

e)
```
void producto( void )
{
  int a, b, c, resultado;
  printf( "Introduzca tres enteros: " )
  scanf( "%d%d%d", &a, &b, &c );
  resultado = a * b * c;
  printf( "El resultado es %d", resultado );
  return resultado;
}
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**5.1** a) Función. b) Llamada a función. c) Variable local. d) **return**. e) **void**. f) Alcance. g) **return;** o **return expresion;** o al encontrar la llave derecha de fin de función. h) Prototipo de función. i) **rand**. j) **srand**. k) **auto**, **register**, **extern**, **static**. l) **auto**. m) **register**. n) Externa, global. o) **static**. p) Alcance de función, alcance de archivo, alcance de bloque, alcance de prototipo de función. q) Recursividad. r) Base.

**5.2** a) Alcance de bloque. b) Alcance de bloque. c) Alcance de archivo. d) Alcance de archivo. e) Alcance de archivo. f) Alcance de prototipo de función.

**5.3**

```
/* ej05_03.c */
/* Verificación de las funciones matemáticas de la biblioteca */
#include <stdio.h>
#include <math.h>

/* la función main inicia la ejecución del programa */
int main()
{
   /* calcula y despliega la raiz cuadrada */
   printf( "sqrt(%.1f) = %.1f\n", 900.0, sqrt( 900.0 ) );
   printf( "sqrt(%.1f) = %.1f\n", 9.0, sqrt( 9.0 ) );

   /* calcula y despliega la función exponencial e a la x */
   printf( "exp(%.1f) = %f\n", 1.0, exp( 1.0 ) );
   printf( "exp(%.1f) = %f\n", 2.0, exp( 2.0 ) );

   /* calcula y despliega el logaritmo (base e) */
   printf( "log(%f) = %.1f\n", 2.718282, log( 2.718282 ) );
   printf( "log(%f) = %.1f\n", 7.389056, log( 7.389056 ) );

   /* calcula y despliega el logaritmo (base 10) */
   printf( "log10(%.1f) = %.1f\n", 1.0, log10( 1.0 ) );
   printf( "log10(%.1f) = %.1f\n", 10.0, log10( 10.0 ) );
   printf( "log10(%.1f) = %.1f\n", 100.0, log10( 100.0 ) );

   /* calcula y despliega el valor absoluto */
   printf( "fabs(%.1f) = %.1f\n", 13.5, fabs( 13.5 ) );
   printf( "fabs(%.1f) = %.1f\n", 0.0, fabs( 0.0 ) );
   printf( "fabs(%.1f) = %.1f\n", -13.5, fabs( -13.5 ) );

   /* calcula y despliega ceil( x ) */
   printf( "ceil(%.1f) = %.1f\n", 9.2, ceil( 9.2 ) );
   printf( "ceil(%.1f) = %.1f\n", -9.8, ceil( -9.8 ) );

   /* calcula y despliega floor( x ) */
```

(*continúa en la siguiente página*)

```
    printf( “floor(%.1f) = %.1f\n”, 9.2, floor( 9.2 ) );
    printf( “floor(%.1f) = %.1f\n”, -9.8, floor( -9.8 ) );

    /* calcula y despliega pow( x, y ) */
    printf( “pow(%.1f, %.1f) = %.1f\n”, 2.0, 7.0, pow( 2.0, 7.0 ) );
    printf( “pow(%.1f, %.1f) = %.1f\n”, 9.0, 0.5, pow( 9.0, 0.5 ) );

    /* calcula y despliega fmod( x, y ) */
    printf( “fmod(%.3f/%.3f) = %.3f\n”, 13.675, 2.333,
             fmod( 13.675, 2.333 ) );

    /* calcula y despliega sin( x ) */
    printf( “sin(%.1f) = %.1f\n”, 0.0, sin( 0.0 ) );

    /* calcula y despliega cos( x ) */
    printf( “cos(%.1f) = %.1f\n”, 0.0, cos( 0.0 ) );

    /* calcula y despliega tan( x ) */
    printf( “tan(%.1f) = %.1f\n”, 0.0, tan( 0.0 ) );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675/2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

**5.4** a) `double hipotenusa( double lado1, double lado2 )`
b) `int elMenor( int x, int y, int z )`
c) `void instrucciones( void )`
d) `float intAfloat ( int numero )`

**5.5** a) `double hipotenusa( double lado1, double lado2 );`
b) `int elMenor( int x, int y, int z );`
c) `void instrucciones( void );`
d) `float intAfloat ( int numero );`

**5.6** a) `register int cuenta  = 0;`
b) `static float ultVal;`
c) `static int numero;`
[*Nota:* Esto podría aparecer fuera de cualquier definición de función.]

**5.7** a) Error: la función **`h`** está definida en la función **`g`**.
Corrección: mueva la definición de **`h`** fuera de la definición de **`g`**.
b) Error: se supone que el cuerpo de la función debe retornar un entero, pero no lo hace.
Corrección: elimine la variable **`resultado`** y coloque la siguiente instrucción en la función.

```
return x + y;
```

c) Error: no devuelve el resultado de **`n + suma(n -1 );`** **`suma`** devuelve un resultado incorrecto.
Corrección: rescriba la instrucción en la cláusula else como

```
return n + suma( n – 1 );
```

d) Error: el punto y coma después del paréntesis derecho que encierra la lista de parámetros, y la redefinición del parámetro **`a`** en la definición de la función.
Corrección: elimine el punto y coma después del paréntesis derecho de la lista de parámetros, y elimine la declaración **`float a`**; del cuerpo de la función.
e) Error: la función devuelve un valor cuando no debería.
Corrección: elimine la instrucción **`return`**.

## EJERCICIOS

**5.8** Muestre el valor de **`x`** después de que se realice cada una de las siguientes instrucciones.
a) **`x = fabs( 7.5 );`**
b) **`x = floor ( 7.5 );`**
c) **`x = fabs( 0.0 );`**
d) **`x = ceil( 0.0 );`**
e) **`x = fabs( -6.4 );`**
f) **`x = ceil( -6.4 );`**
g) **`x = ceil ( -fabs( -8 + floor( -5.5 ) ) );`**

**5.9** Un estacionamiento cobra la cuota mínima de $2.00 por las tres primeras horas de estacionamiento. El estacionamiento cobra $0.50 adicional por hora o *fracción* después del tiempo mínimo. El cobro máximo para cualquier periodo de 24 horas es de $10.00. Suponga que ningún automóvil se estaciona por más de 24 horas, al mismo tiempo que otro. Escriba un programa que calcule e imprima los cobros por cada uno de los tres clientes que se estacionaron ayer en el estacionamiento. Debe introducir el número de horas que cada cliente pasó estacionado ahí. Su programa debe imprimir los resultados en una forma tabular, y debe calcular e imprimir los recibos de las percepciones de ayer. El programa debe utilizar la función **`calculaImporte`** para determinar el importe de cada cliente. Sus salidas deben ser semejantes al formato siguiente:

```
Automóvil      Horas    Importe
1                1.5       2.00
2                4.0       2.50
3               24.0      10.00
Total 29.5     14.50
```

**5.10** Una aplicación de la función **`floor`** es la de redondear un valor al entero más cercano. La instrucción:

```
y = floor(x + .5 );
```

redondea el número **`x`** al entero más cercano, y asigna el resultado a **`y`**. Escriba un programa que lea varios números y utilice la instrucción anterior para redondear estos números al entero más cercano. Por cada uno de los números procesados, imprima el número original y el número redondeado.

**5.11** La función **`floor`** puede utilizarse para redondear un número a una posición decimal determinada. La instrucción:

```
y = floor( x * 10 + .5 ) / 10;
```

redondea **x** a la posición de las décimas (la primera posición a la derecha del punto decimal). La instrucción

```
y = floor( x * 10 + .5 ) / 100;
```

redondea **x** a la posición de las centésimas (la segunda posición a la derecha del punto decimal). Escriba un programa que defina cuatro funciones para redondear al número **x** de distintas maneras:

a) **redondeaAentero( numero )**
b) **redondeaAdecimas ( numero )**
c) **redondeaAcentesimas ( numero )**
d) **redondeaAmilesimas ( numero )**

Por cada valor leído, su programa debe imprimir el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana, y el número redondeado a la milésima más cercana.

**5.12** Responda cada una de las siguientes preguntas.
a) ¿Qué significa elegir números de manera "aleatoria"?
b) ¿Por qué la función **rand** es tan útil para simular juegos de azar?
c) ¿Por qué randomiza un programa por medio de **srand**? ¿Bajo qué circunstancias es recomendable no randomizar?
d) ¿Por qué a menudo es necesario escalar y/o modificar los valores producidos por **rand**?
e) ¿Por qué la simulación computarizada de situaciones reales es una técnica muy útil?

**5.13** Escriba instrucciones que asignen enteros de manera aleatoria a la variable $n$ en los siguientes rangos:
a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

**5.14** Para cada uno de los siguientes conjuntos de enteros, escriba una instrucción individual que imprima un número aleatorio de los conjuntos.
a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14,18, 22.

**5.15** Defina una función llamada **hipotenusa** que calcule la longitud de la hipotenusa de un triángulo recto, cuando se introducen los otros dos lados. Utilice esta función en un programa que determine la longitud de la hipotenusa para cada uno de los siguientes triángulos. La función debe tomar dos argumentos de tipo **double** y devolver la hipotenusa como **double**. Verifique su programa con los valores de los lados especificados en la figura 5.18.

| Triángulo | Lado 1 | Lado 2 |
|---|---|---|
| 1 | 3.0 | 4.0 |
| 2 | 5.0 | 12.0 |
| 3 | 8.0 | 15.0 |

**Figura 5.18** Valores de ejemplo para los lados del triángulo para el ejercicio 5.15.

**5.16** Escriba una función **potenciaInt( base, exponente)** que devuelva el valor de:

$\texttt{base}^{\texttt{exponente}}$

Por ejemplo, **potenciaInt(3, 4)=3*3*3*3**. Suponga que **exponente** es un entero positivo diferente de cero, y **base** es un entero. La función **potenciaInt** debe utilizar **for** para controlar los cálculos. No utilice las funciones matemáticas de la biblioteca.

**5.17** Escriba una función **multiplo** que determine para un par de enteros, si el segundo es múltiplo del primero. La función debe tomar dos argumentos enteros y devolver **1** (verdadero) si el segundo es un múltiplo del primero, y de lo contrario **0** (falso). Utilice esta función en un programa que introduzca una serie de pares de enteros.

**5.18** Escriba un programa que introduzca una serie de enteros y los pase, uno a la vez, a una función llamada **impar** que utilice el operador módulo para determinar si un entero es impar. La función debe tomar un argumento entero y devolver **1** si el entero es impar o **0** si no lo es.

**5.19** Escriba una función que despliegue en el margen izquierdo de la pantalla un cuadrado sólido de asteriscos cuyas medidas se especifiquen mediante el parámetro **lado**. Por ejemplo, si **lado** es **4**, la función despliega:

```
****
****
****
****
```

**5.20** Modifique la función creada en el ejercicio 5.19 para formar el cuadrado de cualquier carácter que especifiquemos en el parámetro **caracterLlenado**. De este modo, si lado es igual a **5** y **caracterLlenado** es "**#**", entonces esta función debe imprimir:

```
#####
#####
#####
#####
#####
```

**5.21** Utilice técnicas similares a las empleadas en los ejercicios 5.19 y 5.20 para producir un programa que grafique un número variado de figuras.

**5.22** Escriba segmentos de programa que lleven a cabo cada una de las siguientes tareas:
a) Calcular la parte entera de un cociente, cuando el entero **a** se divide entre el entero **b**.
b) Calcular el residuo entero, cuando el entero a se divide entre el entero **b**.
c) Utilice los segmentos de programa desarrolladas en a) y b), para escribir una función que introduzca un entero entre **1** y **32767** y que imprima una serie de dígitos, y que cada par de ellos se encuentre separado por dos espacios. Por ejemplo, el entero **4562** se debe imprimir como:

```
4  5  6  2
```

**5.23** Escriba una función que tome el tiempo en tres argumentos enteros (para horas, minutos, y segundos), y que devuelva el número de segundos desde la última vez que el reloj "marcó las 12". Utilice esta función para calcular los segundos que existen entre dos horas, las cuales se miden con el ciclo de 12 horas del reloj.

**5.24** Implemente las siguientes funciones enteras:
a) La función **celsius** devuelve el equivalente en Celsius de la temperatura en Fahrenheit.
b) La función **fahrenheit** devuelve el equivalente en Fahrenheit de la temperatura en Celsius.
c) Utilice estas funciones para escribir un programa que imprima una gráfica que muestre el equivalente en Fahrenheit de las temperaturas Celsius de 0 a 100 grados, y los equivalentes Celsius de todas las temperaturas Fahrenheit de 32 a 212 grados. Imprima las salidas de forma tabular de modo que minimice el número de líneas de salida, pero que sean claras.

**5.25** Escriba una función que devuelva el más pequeño de tres números de punto flotante.

**5.26** Se dice que un número entero es un *número perfecto,* si la suma de sus factores, incluso el 1 (pero no el número mismo), arroja el mismo número. Por ejemplo, 6 es un número perfecto debido a que 6 = 1 + 2 + 3. Escriba la función perfecto que determine si el parámetro numero es un número perfecto. Utilice esta función en un programa que determine e imprima los números perfectos entre 1 y 1000. Imprima los factores de cada número perfecto para confirmar que el número es realmente perfecto. Rete el poder de su computadora y pruebe números más grandes que 1000.

**5.27** Se dice que un entero es *primo* si sólo es divisible entre 1 y entre sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no lo son.
a) Escriba una función que determine si un número es primo.
b) Utilice esta función en un programa que determine e imprima todos los números primos entre 1 y 10,000. ¿Cuántos de estos 10,000 números tiene que verificar realmente antes de que esté seguro de que encontró todos los números primos?

c) Inicialmente podría usted pensar que $n/2$ es el límite máximo que debe probar para ver si un número es primo, pero sólo necesita ir tan arriba como la raíz cuadrada de $n$. ¿Por qué? Rescriba el programa, y ejecútelo de ambas maneras. Estime la mejora en el rendimiento.

**5.28** Escriba una función que tome un valor entero y devuelva el número con los dígitos invertidos. Por ejemplo, dado el número 7631, la función debe regresar 1367.

**5.29** El *máximo común divisor (MCD)* de dos enteros es el entero más grande que divide cada uno de los números. Escriba un programa **`mcd`** que devuelva el máximo común divisor de dos enteros.

**5.30** Escriba una función **`puntosCalidad`** que tome el promedio de un estudiante y devuelva 4 si el promedio del estudiante está entre 90-100, 3 si el promedio es 80-89, 2 si el promedio es 70-79, 1 si el promedio es 60-69, y 0 si el promedio es menor que 60.

**5.31** Escriba un programa que simule un volado (el lanzamiento de una moneda). Por cada volado, el programa deberá imprimir **`Cara`** o **`Cruz`**. Permita que el programa lance la moneda 100 veces y cuente el número de veces que aparece cada lado de la moneda. Imprima los resultados. El programa debe llamar a una función **`aparte`** llamada resultado, la cual no tiene argumentos y devuelve **`0`** para Cara y **`1`** para Cruz. [*Nota:* Si el programa simula de manera realista el lanzamiento de monedas, entonces cada lado debe aparecer aproximadamente la mitad de las veces, para un total de 50 Caras y 50 Cruces.]

**5.32** Las computadoras juegan un rol cada vez más importante en la educación. Escriba un programa que ayude a cualquier estudiante de primaria a aprender a multiplicar. Utilice **`rand`** para producir dos enteros positivos de dos dígitos. Después, debe escribir una pregunta como ésta:

**`¿Cuánto es 6 por 7?`**

Entonces, el estudiante escribe la respuesta. Su programa verifica la respuesta. Si es correcta, imprime "**`¡Muy bien!`**" y hace otra pregunta. Si la pregunta es incorrecta, imprime "**`No. Por favor intenta de nuevo`**", lo que permite al estudiante intentar la misma pregunta de manera repetida hasta que finalmente la contesta correctamente.

**5.33** Al uso de las computadoras en la educación se le conoce como *Educación Asistida por Computadora (EAC)*. Un problema que se presenta en los ambientes EAC es la fatiga estudiantil. Esto puede eliminarse variando los diálogos de las computadoras para mantener la atención de los estudiantes. Modifique el programa del ejercicio 5.32, de manera que se impriman distintos comentarios para cada pregunta contestada de manera correcta y para cada pregunta contestada de manera incorrecta, de la siguiente forma:

Mensajes para una respuesta correcta:

```
¡Muy bien!
¡Excelente!
¡Buen trabajo!
¡Mantén ese buen rendimiento!
```

Mensajes para una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Trata una vez más.
No te rindas!
No. Sigue intentando.
```

Utilice el generador de números aleatorios para elegir un número de 1 a 4 y que seleccione el mensaje apropiado para cada respuesta. Utilice una instrucción **`switch`** con instrucciones **`printf`** para configurar los mensajes.

**5.34** Sistemas más sofisticados de educación asistida por computadora monitorean el rendimiento de un estudiante a lo largo de un periodo de tiempo. A menudo, la decisión de comenzar un nuevo tema se basa en el éxito del estudiante con los temas previos. Modifique el programa del ejercicio 5.33 para que cuente el número de respuestas correctas e incorrectas del estudiante. Después de contestar diez preguntas, su programa debe calcular el porcentaje de respuestas correctas. Si el porcentaje es menor que 75 por ciento, su programa debe imprimir "**`Por favor, pide ayuda adicional a tu profesor`**" y terminar.

**5.35** Escriba un programa en C que juegue el juego de "adivina un número" de la siguiente manera: su programa elige un número que debe adivinar el usuario, seleccionando al azar un número entero en el rango de 1 a 1000. Entonces, el programa escribe:

```
Tengo un número entre 1 y 1000
Puedes adivinar cuál es?
Por favor escribe tu primera respuesta
```

El jugador escribe su primera respuesta. El programa responde con uno de los siguientes mensajes:

```
1. ¡Excelente! ¡Adivinaste el número!
   Quieres jugar otra vez (¿s o n?)
2. Muy abajo. Intenta de nuevo.
3. Muy arriba. Intenta de nuevo.
```

Si la respuesta del jugador es incorrecta, su programa debe entrar en un ciclo hasta que finalmente el jugador adivine el número correcto. Su programa debe continuar indicándole al jugador **`Muy bajo`** o **`Muy alto`**, para ayudarle a "acercarse" a la respuesta correcta. [*Nota:* La técnica de búsqueda empleada en este problema se conoce como búsqueda binaria. Hablaremos más acerca de esto en el siguiente problema.]

**5.36** Modifique el programa del ejercicio 5.35 para contar el número de respuestas correctas que escribió el usuario. Si el número es 10 o menos, imprima ¡**`O sabe el secreto, o tiene suerte`**! Si el jugador adivina el número en diez ocasiones, entonces imprima ¡Ajá! ¡**`Usted sabe el secreto`**! Si el jugador necesita más de 10 intentos, entonces imprima ¡**`Usted puede hacerlo mejor`**! ¿Por qué no debe llevarse más de diez intentos? Bien, con cada "respuesta buena" el jugador debería ser capaz de eliminar la mitad de los números. Ahora sabe por qué cualquier número de 1 a 1000 se puede adivinar en 10 o menos oportunidades.

**5.37** Escriba una función recursiva **`potencia( base, exponente )`** que cuando se invoque devuelva

$\texttt{base}^{\texttt{exponente}}$

Por ejemplo, **`potencia ( 3, 4 )= 3 * 3 * 3 * 3`**. Suponga que *exponente* es un entero mayor o igual que 1. *Pista:* El paso recursivo utilizará la relación:

$\texttt{base}^{\texttt{exponente}} = \texttt{base} * \texttt{base}^{\texttt{exponente - 1}}$

y la condición de terminación ocurre cuando **`exponente`** es igual a 1 debido a que

$\texttt{base}^{1} = \texttt{base}$

**5.38** La serie de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, …

comienza con los términos 0 y 11, y tiene la propiedad de que cada término sucesivo es la suma de los dos términos precedentes. a) Escriba una función *no recursiva* **`fibonacci(n)`** que calcule el enésimo número de Fibonacci. b) Determine el número de Fibonacci más grande que se puede imprimir en su sistema. Modifique el programa del inciso a) para utilizar un número **`double`** en lugar de un **`int`** para calcular y devolver los números de Fibonacci. Permita que el programa haga un ciclo hasta que falle debido a que excede el valor más alto.

**5.39** (*Las torres de Hanoi.*) Todo científico de computación en ciernes debe luchar con cierta clase de problemas, y la Torres de Hanoi (vea la figura 5.19) es uno de los más famosos. La leyenda cuenta que en un templo del lejano oriente, los sacerdotes intentaban mover una pila de discos de un asta a otra. El asta inicial contenía 64 discos ensartados y ordenados de abajo hacia arriba en orden de tamaño decreciente. Los sacerdotes intentaban mover la pila de una primera a una segunda bajo las restricciones de que sólo podían mover un disco a la vez, y en ningún

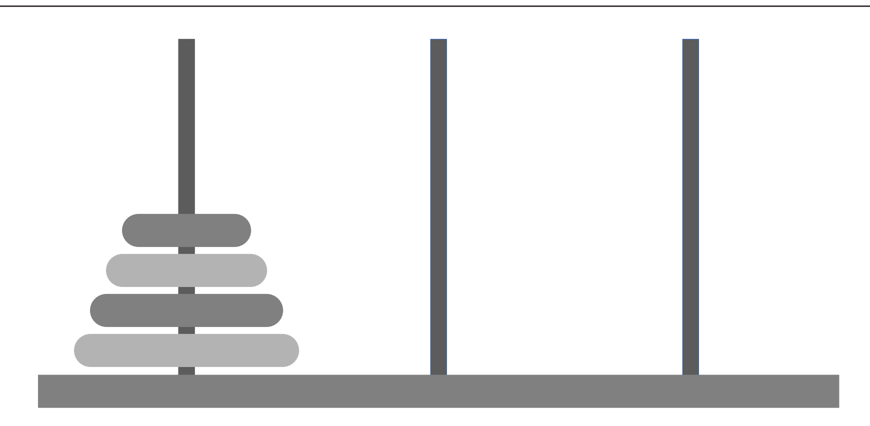


**Figura 5.19** Las Torres de Hanoi para el caso de cuatro discos.

momento podían colocar un disco más grande arriba de uno más pequeño. Una tercera asta estaba disponible para almacenar los discos de manera temporal. Supuestamente el mundo se acabará cuando los sacerdotes completen su tarea, por lo que tenemos pocos motivos para facilitar sus esfuerzos.

Vamos a suponer que los sacerdotes intentan mover los discos del asta 1 al asta 3. Queremos desarrollar un algoritmo que imprima la secuencia precisa de la transferencia de cada disco.

Si quisiéramos afrontar este problema con métodos tradicionales, rápidamente nos encontraríamos atascados sin esperanza para manejar los discos. En lugar de esto, si atacamos el problema considerando la recursividad, la tarea se hace posible automáticamente. Podemos considerar mover los *n* discos en términos del movimiento de sólo *n*-1 discos (y por ende la recursividad) de la siguiente manera:

a) Mueva $n - 1$ discos del asta 1 al asta 2, utilice el asta 3 como área de almacenamiento temporal.
b) Mueva el último disco (el mayor) del asta 1 al asta 3.
c) Mueva los $n - 1$, del asta 2 al asta 3, utilizando el asta 1 como área de almacenamiento temporal.

El proceso finaliza cuando la última tarea involucra el movimiento del disco $n - 1$. Es decir, el caso base. Esto se lleva a cabo mediante la tarea trivial de mover un disco, sin la necesidad del área de almacenamiento temporal.

Escriba un programa para resolver el problema de las Torres de Hanoi. Utilice una función recursiva con cuatro parámetros:

a) El número de discos a mover.
b) El asta en la que se encuentran ensartados los discos.
c) El asta a la que se moverán los discos.
d) El asta que se utilizará como área de almacenamiento temporal.

Su programa debe imprimir las instrucciones precisas necesarias para mover los discos desde el asta inicial al asta de destino. Por ejemplo, para mover una pila con tres discos del asta 1 al asta 3, su programa debe imprimir la siguiente serie de movimientos:

1 → 3 (Esto significa mover un disco del asta 1 al asta 3).
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

**5.40** Cualquier programa que puede implementarse de manera recursiva, puede implementarse de manera iterativa, sin embargo, en ocasiones con una considerable dificultad y menor caridad. Intente escribir una versión iterativa de las Torres de Hanoi. Si tiene éxito, compare su versión iterativa con la versión recursiva desarrollada en el ejercicio 5.39. Investigue los problemas de rendimiento, claridad, y su habilidad para demostrar la eficiencia de los programas.

**5.41** (*Cómo visualizar la recursividad.*) Es interesante observar en acción "a la recursividad". Modifique la función factorial de la figura 5.14 para imprimir su variable local y su llamada recursiva a la función. Para cada llamada recursiva, despliegue las salidas en una línea separada y agregue un nivel de sangrado. Haga lo mejor posible por hacer sus salidas claras, interesantes, y significativas. Aquí, su meta es diseñar e implementar un formato de salida que ayude a una persona a entender mejor la recursividad. Usted podría querer incluir dichas capacidades gráficas a los muchos otros ejemplos y ejercicios que aparecen a través del libro.

**5.42** El máximo común divisor de los enteros `x` y `y` es el entero más grande que divide tanto a `x` como a `y`. Escriba una función recursiva `mcd` que devuelva el máximo común divisor de `x` y `y`. El `mcd` de `x` y `y` se define de manera recursiva de la siguiente manera: si `y` es igual a `0`, entonces `mcd(x, y)` es `x`, de lo contrario `mcd(( x, y )` es `mcd( y, x%y )`, en donde `%` es el operador módulo.

**5.43** ¿Será posible llamar a `main` de manera recursiva? Escriba un programa que contenga una función `main`. Incluya la variable `static` cuenta, inicializada en 1. Postincremente e imprima el valor de cuenta cada vez que se invoque a `main`. Ejecute su programa. ¿Qué sucede?

**5.44** Los ejercicios 5.32 a 5.34 desarrollaron un programa de educación asistida por computadora para enseñar a un estudiante de primaria a multiplicar. Este ejercicio sugiere mejoras a ese programa.

a) Modifique el programa para permitir al usuario registrar el nivel de capacidad. Un nivel igual a 1 significa el uso de números de un solo dígito para el problema, un nivel igual a dos significa el uso de números de dos dígitos, etcétera.
b) Modifique el programa para permitir al usuario elegir el tipo de problemas que desea estudiar. Una opción igual a 1 significa sólo problemas de sumas, 2 significa sólo problemas de restas, 3 significa sólo problemas de multiplicación, 4 significa sólo problemas de división, y 5 significa la mezcla aleatoria de problemas de todos los tipos.

**5.45** Escriba una función **`distancia`** que calcule la distancia entre dos puntos (*x1*, *y1*) y (*x2*, *y2*). Todos los números y los valores de retorno deben ser de tipo **`double`**.

**5.46** ¿Qué hace el siguiente programa?

```
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int c; /* variable para mantener el carácter introducido por el usuario */

   if ( ( c = getchar() ) != EOF ) {
      main();
      printf( “%c”, c );
   } /* fin de if */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**5.47** ¿Qué hace el siguiente programa?

```
#include <stdio.h>

int misterio( int a, int b ); /* prototipo de función */

/* la función main comienza la ejecución del programa */
int main()
{
   int x; /* primer entero */
   int y; /* segundo entero */

   printf( “Introduzca dos enteros: “ );
   scanf( “%d%d”, &x, &y );

   printf( “El resultado es %d\n”, misterio( x, y ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* El parámetro b debe ser un entero positivo
   para evitar la recursividad infinita */
int misterio( int a, int b )
{
   /* caso base */
   if ( b == 1 ) {
      return a;
   } /* fin de if */
   else { /* paso recursivo */
      return a + misterio( a, b - 1 );
   } /* fin de else */

} /* fin de la función misterio */
```

**5.48** Después de que haya determinado lo que hace el programa de la figura 5.47, modifíquelo para que funcione de manera apropiada después de eliminar la restricción de que el segundo argumento sea positivo.

**5.49** Escriba un programa que verifique tantas funciones matemáticas de la biblioteca de la figura 5.2 como pueda. Practique con cada una de estas funciones haciendo que su programa imprima las tablas de retorno de los valores para una diversidad de valores de argumentos.

**5.50** Encuentre el error en cada uno de los siguientes segmentos de programa y explique cómo corregirlos:

a)
```
double cubo( float ); /* Prototipo de función */
...
cubo (float numero )  /* definición de función */
{
   return numero = numero * numero;
}
```
b)
```
register auto int x = 7;
```
c)
```
int numeroAleatorio = srand();
```
d)
```
double y = 123.45678;
int x;
x = y;
printf( "%f\n", (double) x );
```
e)
```
double cuadrado( double numero )
{
   double numero;
   return numero * numero;
}
```
f)
```
int suma( int n )
{
if ( n == 0 )
   return 0;
else
   return n + suma( n );
}
```

**5.51** Modifique el programa del juego de craps que aparece en la figura 5.10 para permitir las apuestas. Empaque, como una función, la porción del programa que ejecuta un juego de craps. Inicialice la variable `saldoBanco` en $1000. Indique al usuario que introduzca la `apuesta`. Utilice un ciclo `while` para verificar si la `apuesta` es menor o igual que `saldoBanco`; si no es así, indique al usuario que reintroduzca la apuesta hasta que lo haga con una cantidad válida. Después de introducir una cantidad válida, ejecute el juego de craps. Si el jugador pierde, disminuya `saldoBanco` con el importe de la apuesta, imprima el nuevo `saldoBanco`, verifique si `saldoBanco` es igual que cero, y si lo es imprima el mensaje **"Lo siento. ¡Su saldo se agoto!"** Durante el transcurso del juego, imprima mensajes para crear algo de "conversación" tal como **"mhm…, parece que va a la quiebra"**, o **"¡Ande!, atrévase!"**, o **"¡Ya estás grande, ahora es el momento de arriesgarse!"**

# 6

# Arreglos en C

## Objetivos

- Introducir la estructura de datos tipo arreglo.
- Comprender el uso de arreglos para almacenar, ordenar y buscar listas y tablas de valores.
- Aprender cómo declarar e inicializar un arreglo, y cómo hacer referencia a elementos individuales de un arreglo.
- Entender como pasar arreglos a funciones.
- Comprender las técnicas básicas de ordenamiento.
- Declarar y manipular arreglos con múltiples subíndices.




*Entre sollozos y lágrimas descubrió*
*A aquellos de mayor tamaño…*
Lewis Carroll

*Intenta hasta el final, y no te detengas ante la duda;*
*Nada es tan difícil, la búsqueda lo demostrará.*
Robert Herrick

*Ahora ve, escríbelo antes que nadie en una tabla,*
*y anótalo en un libro.*
Isaías 30:8

*Está guardado en mi memoria,*
*y tú deberás guardar la llave.*
William Shakespeare

## Plan general



## 6.1 Introducción

Este capítulo sirve como una introducción al importante tema de las estructuras de datos. Los *arreglos* son estructuras de datos que consisten en elementos de datos relacionados del mismo tipo. En el capítulo 10, explicaremos la noción de C de **struct** (estructura), una estructura de datos que consiste en elementos de datos relacionados que posiblemente sean de diferentes tipos. Los arreglos y las estructuras son entidades "estáticas" que mantienen el mismo tamaño durante la ejecución del programa (por supuesto, podrían pertenecer a la clase de almacenamiento automático y, por lo tanto, crearse y destruirse cada vez que se entra y se sale de los bloques en los que se definen). En el capítulo 12, presentaremos estructuras de datos dinámicas como listas, colas, pilas y árboles que pueden crecer y disminuir durante la ejecución de los programas.

## 6.2 Arreglos

Un arreglo es un grupo consecutivo de localidades de memoria relacionadas por el hecho de que tienen el mismo nombre y el mismo tipo. Para hacer referencia a una localidad o a un elemento del arreglo en particular, especificamos el nombre del arreglo y la *posición numérica* del elemento en particular dentro del arreglo.

La figura 6.1 muestra un arreglo de enteros llamado **c**. Este arreglo contiene 12 *elementos*. Es posible hacer referencia a cualquiera de estos elementos al dar el nombre del arreglo seguido por la posición numérica del elemento en particular dentro de corchetes (**[]**). El primer elemento de cada arreglo es el *elemento cero*. Entonces, la referencia al primer elemento del arreglo **c** es **c[ 0 ]**, la referencia al segundo elemento del arreglo es **c[ 1 ]**, la referencia al séptimo elemento del arreglo es **c[ 6 ]**, y en general, la referencia al iésimo elemento del arreglo **c** es **c[ i – 1 ]**. Los nombres de arreglo, como los demás nombres de variables, pueden contener sólo letras, dígitos y guiones bajos. Los nombres de arreglos no pueden comenzar con un dígito.

La posición numérica que se encuentra entre corchetes se denomina, de manera formal, *subíndice*. Un subíndice debe ser un entero o una expresión entera. Si un programa utiliza una expresión como subíndice, entonces la expresión es evaluada para determinar el subíndice. Por ejemplo, si **a = 5** y **b = 6**, entonces la instrucción

```
c[ a + b ] +=2;
```

suma 2 al elemento **c[ 11 ]**. Observe que el subíndice del nombre del arreglo es un lvalue; éste sólo puede utilizarse del lado izquierdo de la asignación.

Examinemos con más detalle el arreglo **c** de la figura 6.1. El *nombre* de todo el arreglo es **c**. A sus **12** elementos se hace referencia como **c[ 0 ]**, **c[ 1 ]**, **c[ 2 ]**,…, **c[ 11 ]**. El *valor* almacenado en **c[ 0 ]** es **-45**, el valor de **c[ 1 ]** es **6**, el valor de **c[ 2 ]** es **0**, el valor de **c[ 7 ]** es **62** y el valor de **c[ 11 ]** es **78**. Para desplegar la suma de los valores que se encuentran en las primeras tres posiciones del arreglo **c**, escribiríamos

```
printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );
```

Nombre del arreglo (observe que todos los elementos de este arreglo tienen el mismo nombre, **c**)

| | |
|---|---|
| c [ 0 ] | -45 |
| c [ 1 ] | 6 |
| c [ 2 ] | 0 |
| c [ 3 ] | 72 |
| c [ 4 ] | 1543 |
| c [ 5 ] | -89 |
| c [ 6 ] | 0 |
| c [ 7 ] | 62 |
| c [ 8 ] | -3 |
| c [ 9 ] | 1 |
| c [ 10 ] | 6453 |
| c [ 11 ] | 78 |

Posición numérica del elemento dentro del arreglo **c**

**Figura 6.1** Arreglo de 12 elementos.

Para dividir entre **2** el valor del séptimo elemento del arreglo **c**, y asignar el resultado a la variable **x**, escribiríamos

```
x = c[ 6 ] / 2;
```



**Error común de programación 6.1**

*Es importante notar la diferencia entre el "séptimo elemento del arreglo" y el "elemento siete del arreglo". Debido a que los subíndices de los arreglos comienzan en 0, el "séptimo elemento del arreglo" tiene un subíndice de 6, mientras que "el elemento siete del arreglo" tiene un subíndice de 7 y, en realidad, es el octavo elemento del arreglo. Ésta es una fuente de "errores de desplazamiento en uno".*

Los corchetes que se utilizan para encerrar el subíndice de un arreglo, en realidad se consideran como un operador de C. Los corchetes tienen el mismo nivel de precedencia que el operador de llamadas a función (es decir, el par de paréntesis que se colocan después del nombre de una función para llamar a esa función). La figura 6.2 muestra la precedencia y asociatividad de los operadores que hemos presentado hasta este punto del texto. Éstos aparecen de arriba hacia abajo en orden decreciente de precedencia.

## 6.3 Declaración de arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada arreglo y el número de elementos que necesita el arreglo, de manera que la computadora pueda reservar la cantidad adecuada de memoria. Para indicarle a la computadora que reserve 12 elementos para el arreglo entero **c**, se utiliza la declaración

```
int c[ 12 ];
```

La siguiente declaración

```
int b[ 100 ], x[ 27 ];
```

reserva 100 elementos para el arreglo entero **b**, y 27 elementos para el arreglo entero **x**.

<table>
<tr><th>Operadores</th><th></th><th></th><th></th><th></th><th></th><th>Asociatividad</th><th>Tipo</th></tr>
<tr><td>[ ]</td><td>( )</td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>más alto</td></tr>
<tr><td>++</td><td>--</td><td>!</td><td>(tipo)</td><td></td><td></td><td>derecha a izquierda</td><td>unario</td></tr>
<tr><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td><td>izquierda a derecha</td><td>multiplicación</td></tr>
<tr><td>+</td><td>-</td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>adición</td></tr>
<tr><td><</td><td><=</td><td>></td><td>>=</td><td></td><td></td><td>izquierda a derecha</td><td>de relación</td></tr>
<tr><td>==</td><td>!=</td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>de igualdad</td></tr>
<tr><td>&&</td><td></td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>AND lógico</td></tr>
<tr><td>||</td><td></td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>OR lógico</td></tr>
<tr><td>?:</td><td></td><td></td><td></td><td></td><td></td><td>derecha a izquierda</td><td>condicional</td></tr>
<tr><td>=</td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>derecha a izquierda</td><td>de asignación</td></tr>
<tr><td>,</td><td></td><td></td><td></td><td></td><td></td><td>izquierda a derecha</td><td>coma</td></tr>
</table>

**Figura 6.2** Precedencia y asociatividad de operadores.

Es posible declarar arreglos para que contengan otros tipos de datos. Por ejemplo, un arreglo de tipo **`char`** puede utilizarse para almacenar una cadena de caracteres. En el capítulo 8 explicaremos las cadenas de caracteres y sus similitudes con los arreglos. En el capítulo 7 explicaremos la relación que existe entre los apuntadores y los arreglos.

## 6.4 Ejemplos de arreglos

Esta sección presenta diversos ejemplos que demuestran cómo declarar arreglos, cómo inicializarlos y cómo realizar muchas manipulaciones comunes a ellos.

***Cómo declarar un arreglo y cómo utilizar un ciclo para inicializar sus elementos***

La figura 6.3 utiliza instrucciones **`for`** para inicializar en ceros los elementos de un arreglo entero **`n`** de 10 elementos, y para imprimir dicho arreglo en formato tabular. La primera instrucción **`printf`** (línea 16) despliega la columna de encabezados para las dos columnas impresas en la instrucción for subsiguiente.

```
1  /* Figura 6.3: fig06_03.c
2     inicializar un arreglo */
3  #include <stdio.h>
4
5  /* la función main comienza la ejecución del programa */
6  int main()
7  {
8     int n[ 10 ]; /* n es un arreglo de 10 enteros */
9     int i; /* contador */
10
11    /* inicializa los elementos del arreglo n a 0 del arreglo */
12    for ( i = 0; i < 10; i++ ) {
13       n[ i ] = 0; /* establece el elemento i a 0 */
14    } /* fin de for */
15
16    printf( "%s%13s\n", "Elemento", "Valor" );
17
18    /* muestra el contenido del arreglo n en forma tabular */
```

**Figura 6.3** Inicialización en ceros de los elementos de un arreglo. (Parte 1 de 2.)

```
    for ( i = 0; i < 10; i++ ) {
       printf( "%7d%13d\n", i, n[ i ] );
    } /* fin de for */

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Elemento        Valor
       0            0
       1            0
       2            0
       3            0
       4            0
       5            0
       6            0
       7            0
       8            0
       9            0
```

**Figura 6.3** Inicialización en ceros de los elementos de un arreglo. (Parte 2 de 2.)

***Cómo inicializar un arreglo en una declaración con una lista de inicialización***

Los elementos de un arreglo también pueden inicializarse cuando se declara el arreglo, colocando un signo igual seguido de un par de llaves, **{}**, que contenga una lista de *inicializadores* separados por comas. La figura 6.4 inicializa un arreglo entero de 10 valores (línea 9) y lo imprime en un formato tabular.

Si existen menos inicializadores que elementos en el arreglo, el resto de los elementos del arreglo se inicializa en cero. Por ejemplo, los elementos del arreglo **n** de la figura 6.3 se podrían haber inicializado en cero de la siguiente forma:

```
int n[ 10 ] = { 0 };
```

```
1  /* Figura 6.4: fig06_04.c
2     Inicializa un arreglo con una lista de inicialización */
3  #include <stdio.h>
4
5  /* la función main comienza la ejecución del programa */
6  int main()
7  {
8     /* utiliza la lista de inicialización para inicializar el arreglo n */
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    int i; /* contador */
11
12    printf( "%s%13s\n", "Elemento", "Valor" );
13
14    /* muestra el contenido del arreglo en forma tabular */
15    for ( i = 0; i < 10; i++ ) {
16       printf( "%7d%13d\n", i, n[ i ] );
17    } /* fin de for */
18
19    return 0; /* indica terminación exitosa */
20
21 } /* fin de main */
```

**Figura 6.4** Inicialización de los elementos de un arreglo con una lista de inicialización. (Parte 1 de 2.)

```
Elemento         Valor
       0            32
       1            27
       2            64
       3            18
       4            95
       5            14
       6            90
       7            70
       8            60
       9            37
```

**Figura 6.4** Inicialización de los elementos de un arreglo con una lista de inicialización. (Parte 2 de 2.)

Esto inicializa en cero de manera explícita al primer elemento, y de manera implícita a los nueve elementos restantes, debido a que en el arreglo existen menos inicializadores que elementos. Es importante que recuerde que los arreglos no se inicializan automáticamente en cero. El programador debe inicializar al menos el primer elemento en cero, para que los elementos restantes se inicialicen automáticamente en cero. Este método de inicialización de elementos a cero se lleva a cabo en tiempo de compilación para los arreglos estáticos (**`static`**) y en tiempo de ejecución para los arreglos automáticos.



**Error común de programación 6.2**

*Olvidar inicializar los elementos de un arreglo, cuyos elementos debieran inicializarse.*

La declaración del arreglo

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

ocasiona un error de sintaxis, debido a que hay seis inicializadores y sólo cinco elementos en el arreglo.



**Error común de programación 6.3**

*Proporcionar más inicializadores en una lista de inicialización que elementos en el arreglo, es un error de sintaxis.*

Si se omite el tamaño de un arreglo que se declaró con una lista de inicialización, el número de elementos en el arreglo será el número de elementos de la lista de inicialización. Por ejemplo,

```
int n[] = { 1, 2, 3, 4, 5 };
```

crearía un arreglo de cinco elementos.

### *Cómo especificar el tamaño de un arreglo mediante una constante simbólica y cómo inicializar los elementos de un arreglo mediante cálculos*

La figura 6.5 inicializa los elementos del arreglo de 10 elementos llamado **`s`** con los valores **`2`**, **`4`**, **`6`**, …, **`20`**, y se imprime el arreglo en forma tabular. Los valores se generan multiplicando por **`2`** al contador de ciclo y sumándole **`2`**.

```
/* Figura 6.5: fig06_05.c
   Inicializa los elementos del arreglo s a los enteros pares de 2 a 20 */
#include <stdio.h>
#define TAMANIO 10

/* la función main comienza la ejecución del programa */
int main()
{
```

**Figura 6.5** Generación de valores a colocarse en los elementos de un arreglo. (Parte 1 de 2.)

```
9      /* se puede utilizar la constante simbólica TAMANIO para especificar el
          tamaño del arreglo */
10     int s[ TAMANIO ]; /* el arreglo s contiene 10 elementos */
11     int j; /* conntador */
12
13     for ( j = 0; j < TAMANIO; j++ ) { /* establece los valores */
14        s[ j ] = 2 + 2 * j;
15     } /* fin de for */
16
17     printf( “%s%13s\n”, “Elemento”, “Valor” );
18
19     /* muestra el contenido del arreglo s en forma tabular */
20     for ( j = 0; j < TAMANIO; j++ ) {
21        printf( “%7d%13d\n”, j, s[ j ] );
22     } /* fin de for */
23
24     return 0; /* indica terminación exitosa */
25
26  } /* fin de main */
```

```
Elemento        Valor
       0            2
       1            4
       2            6
       3            8
       4           10
       5           12
       6           14
       7           16
       8           18
       9           20
```

**Figura 6.5** Generación de valores a colocarse en los elementos de un arreglo. (Parte 2 de 2.)

La directiva de preprocesador **#define** se introduce en este programa. La línea 4

```
#define TAMANIO 10
```

define una *constante simbólica* **TAMANIO**, cuyo valor es **10**. Una constante simbólica es un identificador que es reemplazado por el preprocesador de C con *texto de reemplazo*, antes de que se compile el programa. Cuando se preprocesa el programa, todas las ocurrencias de la constante simbólica **TAMANIO** se reemplazan con el texto de reemplazo **10**. Utilizar constantes simbólicas para especificar el tamaño de un arreglo hace que los programas sean más *escalables*. En la figura 6.5, el primer ciclo **for** (línea 13) podría llenar un arreglo de 1000 elementos, si tan solo se modificara el valor de **TAMANIO** en la directiva **#define** de **10** a **1000**. Si no hubiéramos utilizado la constante simbólica **TAMANIO**, hubiéramos tenido que cambiar el programa en tres lugares diferentes para escalarlo y que pudiera manejar un arreglo de 1000 elementos. Esta técnica se vuelve más útil para escribir programas más claros, conforme éstos se vuelven más grandes.



### Error común de programación 6.4

*Finalizar una directiva de preprocesador* **#define** *o* **#include** *con un punto y coma. Recuerde que las directivas de preprocesador no son instrucciones de C.*

Si la directiva de preprocesador **#define** de la línea 4 termina con un punto y coma, el preprocesador reemplaza todas las ocurrencias de la constante simbólica **TAMANIO** con el texto **10**. Esto puede ocasionar errores de sintaxis en tiempo de compilación, o errores de lógica en tiempo de ejecución. Recuerde que el preprocesador no es C; sólo es un manipulador de texto.



### Error común de programación 6.5

*Asignar un valor a una constante simbólica en una instrucción ejecutable, es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio alguno para ella, como lo hace con las variables que contienen valores en tiempo de ejecución.*



### Observación de ingeniería de software 6.1

*Definir el tamaño de un arreglo como una constante simbólica hace que los programas sean más escalables.*



### Buena práctica de programación 6.1

*Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y recuerda al programador que las constantes simbólicas no son variables.*



### Buena práctica de programación 6.2

*En nombres de constantes simbólicas que contengan varias palabras, utilice guiones bajos para separarlas y, así, mejorar su legibilidad.*

***Cómo sumar los elementos de un arreglo***

La figura 6.6 suma los valores contenidos en el arreglo a tipo entero de 12 elementos, **`a`**. El cuerpo de la instrucción **`for`** (línea 16) calcula el total.

***Cómo utilizar arreglos para resumir los resultados de una encuesta***

Nuestro siguiente ejemplo utiliza arreglos para resumir la información recolectada en una encuesta. Considere el enunciado del problema:

> *A cuarenta estudiantes se les preguntó respecto a la calidad de la comida de la cafetería escolar, en una escala de 1 a 10 (1 significa muy mala y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero que resuma los resultados de la encuesta.*

```
1  /* Figura 6.6: fig06_06.c
2     Calcula la suma de los elementos del arreglo */
3  #include <stdio.h>
4    #define TAMANIO 12
5
6  /* la función main comienza la ejecución del programa */
7  int main()
8    {
9     /* utiliza una lista de inicialización para inicializar el arreglo */
10    int a[ TAMANIO ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11    int i; /* contador */
12    int total = 0; /* suma del arreglo */
13
14    /* suma el contenido del arreglo a */
15    for ( i = 0; i < TAMANIO; i++ ) {
16       total += a[ i ];
17    } /* fin de for */
18
19    printf( "El total de los elementos del arreglo es %d\n", total );
20
21    return 0; /* indica terminación exitosa */
22
23 } /* fin de main */
```

```
El total de los elementos del arreglo es 383
```

**Figura 6.6** Cálculo de la suma de los elementos de un arreglo.

Ésta es una aplicación típica de los arreglos (vea la figura 6.7). Nosotros deseamos resumir el número de respuestas de cada tipo (es decir, de 1 a 10). El arreglo **`respuestas`** (línea 17) contiene 40 elementos que almacenan las respuestas de los estudiantes. Utilizamos un arreglo de 11 elementos llamado **`frecuencia`** (línea 14), para contar el número de ocurrencias de cada respuesta. Ignoramos **`frecuencia[ 0 ]`** porque es lógico hacer que la respuesta 1 incremente a **`frecuencia[ 1 ]`**, en lugar de **`frecuencia[ 0 ]`**. Esto nos permite utilizar directamente cada respuesta como el subíndice del arreglo **`frecuencia`**.



### Buena práctica de programación 6.3

*Busque la claridad de los programas. A veces, vale la pena perder un poco de eficiencia en cuanto al uso de la memoria o del procesador, a favor de la creación de programas más claros.*



### Tip de rendimiento 6.1

*En ocasiones, las consideraciones relacionadas con el rendimiento se alejan demasiado de las consideraciones para lograr la claridad.*

```
1  /* Figura 6.7: fig06_07.c
2     Programa de respuestas de examen */
3  #include <stdio.h>
4  #define TAMANIO_RESPUESTA  40 /* define los tamaños de los arreglos */
5  #define TAMANIO_FRECUENCIA 11
6
7  /* la función main comienza la ejecución del programa */
8  int main()
9  {
10    int respuesta; /* contador a través de las 40 respuestas */
11    int rango; /* contador de rangos de 1 a 10 */
12
13    /* inicializa los contadores de frecuancia a 0 */
14    int frecuencia[ TAMANIO_FRECUENCIA ] = { 0 };
15
16    /* coloca las respuestas del exámen dentro del arreglo respuestas */
17    int respuestas[ TAMANIO_RESPUESTA ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18            1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19            5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    /* por cada respuesta, seleciona el valor de un elemento del arreglo
22       respuestas y utiliza dicho valor como subíndice en el arreglo
23       frecuencia para determinar el elemento a incrementar */
24    for ( respuesta = 0; respuesta < TAMANIO_RESPUESTA; respuesta++ ) {
25       ++frecuencia[ respuestas [ respuesta ] ];
26    } /* fin de for */
27
28    /* despliega los resultados */
29    printf( "%s%17s\n", "Rango", "Frecuencia" );
30
31    /* muestra las frecuencias en forma tabular */
32    for ( rango = 1; rango < TAMANIO_FRECUENCIA; rango++ ) {
33       printf( "%6d%17d\n", rango, frecuencia[ rango ] );
34    } /* fin de for */
35
36    return 0; /* indica terminación exitosa */
37
38 } /* fin de main */
```

**Figura 6.7** Programa para analizar una encuesta aplicada a estudiantes. (Parte 1 de 2.)

```
Rango       Frecuencia
    1                2
    2                2
    3                2
    4                2
    5                5
    6               11
    7                5
    8                7
    9                1
   10                3
```

**Figura 6.7** Programa para analizar una encuesta aplicada a estudiantes. (Parte 2 de 2.)

El ciclo **for** (línea 24) toma las respuestas, una a la vez, del arreglo **respuestas** e incrementa uno de los 10 contadores (**frecuencia[ 1 ]** a **frecuencia[ 10 ]**) de dicho arreglo. La instrucción clave del ciclo es la línea 25

```
++frecuencia[ respuestas[ respuesta ] ];
```

la cual incrementa el contador de **frecuencia** adecuado, de acuerdo con el valor de **respuestas[ respuesta ]**. Cuando la variable contador respuesta es **0**, **respuestas[ respuesta ]** es **respuestas[ 0 ]** que es **1**, por lo que la instrucción **++frecuencia[ respuestas[ respuesta ] ]**; en realidad se interpreta como

```
++frecuencia[ 1 ];
```

lo que incrementa el elemento uno del arreglo. Cuando **respuesta** es **1**, **respuestas[ respuesta ]** es **respuestas[ 1 ]** que es **2**, por lo que la instrucción **++frecuencia[ respuestas[ respuesta ] ];** en realidad se interpreta como

```
++frecuencia[ 2 ];
```

lo que incrementa el elemento dos del arreglo. Cuando **respuesta** es **2**, **respuestas[ respuesta ]** es **respuestas[ 2 ]** que es **6**, por lo que la instrucción **++frecuencia[ respuestas[ respuesta ] ];** en realidad se interpreta como

```
++frecuencia[ 6 ];
```

lo que incrementa el elemento seis del arreglo, y así sucesivamente. Observe que independientemente del número de respuestas procesadas en la encuesta, sólo se requiere un arreglo de once elementos (si omitimos el elemento cero) para resumir los resultados. Si la información contuviera valores no permitidos como 13, el programa intentaría agregar **1** a **frecuencia[ 13 ]**. Esto estaría fuera de los límites del arreglo. *C no tiene forma de verificar los límites del arreglo para evitar que la computadora haga referencia a elementos inexistentes del arreglo*. Por lo tanto, un programa en ejecución puede salir o terminar el procesamiento de un arreglo sin advertencia alguna. El programador deberá asegurarse de que todas las referencias a los arreglos permanezcan dentro de estos límites.



**Error común de programación 6.6**

*Hacer referencia a un elemento que se encuentra fuera de los límites del arreglo.*



**Tip para prevenir errores 6.1**

*Cuando se hace un ciclo en torno a un arreglo, el subíndice del arreglo nunca debe ser menor que 0 y siempre debe ser menor que el número total de elementos del arreglo (tamaño –1). Asegúrese que la condición de terminación de ciclo prevenga el acceso de elementos fuera de este rango.*



**Tip para prevenir errores 6.2**

*Los programas deben validar que todos los valores de entrada sean correctos, para evitar que información errónea afecte los cálculos del programa.*

***Cómo graficar los elementos de un arreglo mediante histogramas***

Nuestro siguiente ejemplo (figura 6.8) lee los números de un arreglo y grafica la información en forma de un gráfico de barras o histograma; cada número se imprime, seguido por una barra que consiste en muchos asteriscos. La instrucción anidada **`for`** (línea 20) dibuja las barras. Observe el uso de **`printf("\n")`** para finalizar la barra del histograma (línea 24).

***Cómo tirar un dado 6,000 veces y resumir los resultados en un arreglo***

En el capítulo 5 dijimos que mostraríamos un método más elegante para escribir el programa de dados de la figura 5.8. El problema trata sobre tirar un dado de seis lados 6,000 veces para probar si el generador de números aleatorios realmente producía números aleatorios. La figura 6.9 muestra una versión de este programa con arreglos.

```
1  /* Figura 6.8: fig06_08.c
2     Programa de impresión de un Histograma */
3  #include <stdio.h>
4  #define TAMANIO 10
5
6  /* la función main comienza la ejecución del programa */
7  int main()
8  {
9     /* usar una lista de inicialización para inicializar el arreglo n */
10    int n[ TAMANIO ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
11    int i; /* contador for externo para los  elementos del arreglo */
12    int j; /* contador for interno cuenta *s en cada barra del histograma */
13
14    printf( "%s%13s%17s\n", "Elemento", "Valor", "Histograma" );
15
16    /* para cada elemento del arreglo n, muestra una barra en el histograma */
17    for ( i = 0; i < TAMANIO; i++ ) {
18       printf( "%7d%13d        ", i, n[ i ]) ;
19
20       for ( j = 1; j <= n[ i ]; j++ ) {   /* imprime una barra */
21          printf( "%c", '*' );
22        } /* fin del for interno */
23
24       printf( "\n" ); /* fin de una barra del histograma */
25    } /* fin del for externo */
26
27    return 0; /* indica terminación exitosa */
28
29 } /* fin de main */
```

```
Elemento        Valor        Histograma
      0           19         *******************
      1            3         ***
      2           15         ***************
      3            7         *******
      4           11         ***********
      5            9         *********
      6           13         *************
      7            5         *****
      8           17         *****************
      9            1         *
```

**Figura 6.8** Impresión de un histograma.

```
/* Figura 6.9: fig06_09.c
   Lanza un dado de seis lados 6000 veces */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAMANIO 7

/* la función main comienza la ejecución del programa */
int main()
{
   int cara; /* valor aleatorio del dado entre 1 a 6 */
   int tiro; /* contador de tiros 1 a 6000 */
   int frecuencia[ TAMANIO ] = { 0 }; /* inicializa a cero  la cuenta */

   srand( time( NULL ) ); /* generador de la semilla de números aleatorios */

   /* tira el dado 6000 veces */
   for ( tiro = 1; tiro <= 6000; tiro++ ) {
      cara = 1 + rand() % 6;
      ++frecuencia[ cara ]; /* remplaza la instrucción switch de la línea 26
      de la figura 5.8 */
   } /* fin de for */

   printf( “%s%17s\n”, “Cara”, “Frecuencia” );

   /* muestra los elementos 1-6 de frecuencia en forma tabular */
   for ( cara = 1; cara < TAMANIO; cara++ ) {
      printf( “%4d%17d\n”, cara, frecuencia[ cara ] );
   } /* fin de for */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Cara        Frecuencia
   1              1029
   2               951
   3               987
   4              1033
   5              1010
   6               990
```

**Figura 6.9** Programa de tiro de dados mediante el uso de arreglos en lugar de la instrucción **`switch`**.

***Cómo utilizar arreglos de caracteres para almacenar y manipular cadenas***

Hasta el momento, sólo hemos explicado arreglos enteros. Sin embargo, los arreglos son capaces de almacenar datos de cualquier tipo. Ahora explicaremos el almacenamiento de cadenas en arreglos de caracteres. Hasta este punto, la única capacidad para el procesamiento de cadenas con la que contamos es la impresión de una cadena con **`printf`**. Una cadena como “**`hola`**”, en realidad es un arreglo estático (**`static`**) de caracteres individuales de C.

Los arreglos de caracteres tienen muchas características únicas. Un arreglo de caracteres puede inicializarse mediante una literal de cadena. Por ejemplo,

```
char cadena1[] = “primero”;
```

inicializa los elementos del arreglo **cadena1** mediante los caracteres individuales de la literal de cadena **“primero”**. En este caso, el compilador determina el tamaño del arreglo **cadena1**, basándose en la longitud de la cadena. Es importante observar que la cadena **“primero”** contiene 7 caracteres, *más* un carácter especial de terminación de la cadena llamado *carácter nulo*. Por lo tanto, el arreglo **cadena1** en realidad contiene ocho elementos. La constante que representa el carácter nulo es **‘\0’**. Todas las cadenas en C finalizan con este carácter. Un arreglo de caracteres que representa una cadena siempre debe declararse con el tamaño suficiente para almacenar los caracteres de la cadena y el carácter de terminación nulo.

Los arreglos de caracteres también pueden inicializarse mediante caracteres individuales constantes en una lista de inicialización. La definición anterior es equivalente a

```
char cadena1[] = { ‘p’, ‘r’, ‘i’, ‘m’, ‘e’, ‘r’, ‘o’, ‘\0’ };
```

Debido a que una cadena en realidad es un arreglo de caracteres, podemos acceder directamente a los caracteres individuales de la cadena, utilizando la notación de subíndices. Por ejemplo, **cadena1[ 0 ]** es el carácter **‘p’**, y **cadena1[ 3 ]** es el carácter **‘m’**.

También podemos introducir directamente una cadena en un arreglo de caracteres desde el teclado, utilizando **scanf** y el especificador de conversión **%s**. Por ejemplo,

```
char cadena2[ 20 ];
```

crea un arreglo de caracteres capaz de almacenar una cadena de 19 caracteres y el carácter de terminación nulo. La instrucción

```
scanf( “%s”, cadena2 );
```

lee una cadena introducida desde el teclado en **cadena2**. Observe que el nombre del arreglo pasa a **scanf** sin el **&** que utilizamos para preceder a las variables que no son cadenas. El **&** normalmente se utiliza para proporcionar a **scanf** la ubicación en memoria de una variable, para que ahí pueda almacenarse un valor. En la sección 6.5, cuando expliquemos el paso de arreglos a funciones, veremos que el nombre de un arreglo es la dirección del inicio del arreglo; por lo tanto, el **&** no es necesario.

Es responsabilidad del programador asegurarse de que el arreglo en el que se lee la cadena es capaz de almacenar cualquier cadena que el usuario escriba mediante el teclado. La función **scanf** lee los caracteres introducidos a través del teclado, hasta que encuentra el primer carácter blanco; ésta no verifica el tamaño del arreglo. Por lo tanto, **scanf** puede escribir más allá del final del arreglo.



### Error común de programación 6.7

*No proporcionarle a* ***scanf*** *un arreglo de caracteres lo suficientemente grande para almacenar una cadena escrita mediante el teclado, puede ocasionar la destrucción de los datos de un programa y otros errores en tiempo de ejecución.*

Un arreglo de caracteres que representa a una cadena puede imprimirse con **printf** y el especificador de conversión **%s**. El arreglo **cadena2** se imprime con la instrucción

```
printf( “%s\n”, cadena2 );
```

Observe que **printf**, como **scanf**, no verifica el tamaño del arreglo de caracteres. Los caracteres de la cadena se imprimen hasta que aparece el carácter de terminación nulo.

La figura 6.10 muestra la inicialización de un arreglo de caracteres mediante una literal de cadena, la lectura de una cadena que se encuentra en un arreglo de caracteres, la impresión de un arreglo de caracteres como cadena, y el acceso a los caracteres individuales de la cadena.

```
/* Figura 6.10: fig06_10.c
   Manipulación de arreglos de caracteres como cadenas */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
```

**Figura 6.10** Arreglos de caracteres procesados como cadenas. (Parte 1 de 2.)

```
6  int main()
7  {
8     char cadena1[ 20 ]; /* reserva 20 caracteres */
9     char cadena2[] = "literal de cadena"; /* reserva 18 caracteres */
10    int i; /* contador */
11
12    /* lee la cadena del usuario y la introduce en el arreglo cadena1 */
13    printf("Introduce una cadena: ");
14    scanf( "%s", cadena1 ); /* entrada que finaliza con un espacio en blanco */
15
16    /* muestra las cadenas */
17    printf( "La cadena1 es: %s\ncadena2 es: %s\n"
18            "La cadena1 con espacios entre caracteres es:\n",
19            cadena1, cadena2 );
20
21    /* muestra los caracteres hasta que encuentra el caracter nulo */
22    for ( i = 0; cadena1[ i ] != '\0'; i++ ) {
23       printf( "%c ", cadena1[ i ] );
24    } /* fin de for */
25
26    printf( "\n" );
27
28    return 0; /* indica terminación exitosa */
29
30 } /* fin de main */
```

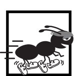
```
Introduce una cadena: Hola amigos
La cadena1 es: Hola
cadena2 es: literal de cadena
La cadena1 con espacios entre caracteres es:
H o l a
```

**Figura 6.10** Arreglos de caracteres procesados como cadenas. (Parte 2 de 2.)

La figura 6.10 utiliza una instrucción **`for`** (línea 22) para generar un ciclo a través del arreglo **`cadena1`** e imprimir los caracteres individuales separados por espacios mediante el especificador de conversión **`%c`**. La condición de la instrucción **`for`**, **`cadena1[ i ] != '\0'`**, es verdadera hasta que el ciclo encuentra el carácter de terminación nulo en la cadena.

***Arreglos estáticos locales y arreglos automáticos locales***

En el capítulo 5 se explicó el especificador de clase de almacenamiento **`static`**. Una variable local **`static`** existe a lo largo de la duración del programa, pero sólo es visible en el cuerpo de la función. Podemos aplicar el especificador **`static`** a la declaración de un arreglo local, para que el arreglo no se genere y se inicialice cada vez que el programa llame a la función, y para que el arreglo no se destruya cada vez que el programa salga de la función. Esto reduce el tiempo de ejecución del programa, en particular de aquellos programas que contienen llamadas frecuentes a funciones que contienen arreglos grandes.



**Tip de rendimiento 6.2**

*En funciones que contienen arreglos automáticos, en donde la función entra y sale con frecuencia del alcance, haga que el arreglo sea* **`static`** *para que éste no se genere cada vez que se invoque a la función.*

Los arreglos **`static`** se inicializan automáticamente en tiempo de compilación. Si el programador no inicializa explícitamente un arreglo **`static`**, el compilador inicializa en cero a los elementos del arreglo.

La figura 6.11 muestra la función **`iniciaArregloEstatico`** (línea 24) con un arreglo local **`static`** (línea 27), y la función **`iniciaArregloAutomatico`** (línea 47) con un arreglo local automático (línea 50).

A la función **`iniciaArregloEstatico`** se le llama dos veces (líneas 12 y 16). El compilador inicializa en cero al arreglo local **`static`** de la función (línea 27). La función imprime el arreglo, le suma 5 a cada elemento y lo imprime nuevamente. La segunda vez que se llama a la función, el arreglo **`static`** contiene los valores almacenados durante la primera llamada a la función. A la función **`iniciaArregloAutomatico`** también se le llama dos veces (líneas 13 y 17). Los elementos del arreglo local automático de la función se inicializan con los valores 1, 2 y 3 (línea 50). La función imprime el arreglo, le suma 5 a cada elemento y lo imprime nuevamente. La segunda vez que se llama a la función, los elementos del arreglo se inicializan nuevamente en 1, 2 y 3, debido a que el arreglo tiene una duración automática de almacenamiento.

### Error común de programación 6.8

*Suponer que los elementos de un arreglo local* **`static`** *se inicializan en cero cada vez que se llama a la función en la que el arreglo está declarado.*

```
1  /* Figura 6.11: fig06_11.c
2     Arreglos estáticos que se inicializan a cero */
3  #include <stdio.h>
4
5  void iniciaArregloEstatico( void );   /* prototipo de la función */
6  void iniciaArregloAutomatico( void ); /* prototipo de la función */
7
8  /* la función main comienza la ejecución del programa */
9  int main()
10 {
11    printf( "Primera llamada a cada funcion:\n" );
12    iniciaArregloEstatico();
13    iniciaArregloAutomatico();
14
15    printf( "\n\nSegunda llamada a cada funcion:\n" );
16    iniciaArregloEstatico();
17    iniciaArregloAutomatico();
18
19    return 0; /* indica terminación exitosa */
20
21 } /* fin de main */
22
23 /* función para demostrar un arreglo estático local */
24 void iniciaArregloEstatico( void )
25 {
26    /* inicializa los elementos a 0 la primera vez que se llama a la función */
27    static int arreglo1[ 3 ];
28    int i; /* contador */
29
30    printf( "\nValores al entrar a iniciaArregloEstatico:\n" );
31
32    /* muestra el contenido del arreglo1 */
33    for ( i = 0; i <= 2; i++ ) {
34       printf( "arreglo1[ %d ] = %d  ", i, arreglo1[ i ] );
35    } /* fin de for */
36
37    printf( "\nValores al salir de iniciaArregloEstatico:\n" );
38
39    /* modifica y muestra el contenido de arreglo1 */
```

**Figura 6.11** Si el programador no inicializa explícitamente los arreglos **`static`**, éstos se inicializan automáticamente en cero. (Parte 1 de 2.)

```
   for ( i = 0; i <= 2; i++ ) {
      printf( "arreglo1[ %d ] = %d  ", i, arreglo1[ i ] += 5 );
   } /* fin de for */

} /* fin de la función iniciaArregloEstatico */

/* función para demostrar un arreglo local automático */
void iniciaArregloAutomatico( void )
{
   /* inicializa los elementos cada vez que se llama a la función */
   int arreglo2[ 3 ] = { 1, 2, 3 };
   int i; /* contador */

   printf( "\n\nValores al entrar a iniciaArregloAutomatico:\n" );

   /* muestra el contenido de arreglo2 */
   for ( i = 0; i <= 2; i++ ) {
      printf("arreglo2[ %d ] = %d  ", i, arreglo2[ i ] );
   } /* fin de for */

   printf( "\nValores al salir de iniciaArregloAutomatico:\n" );

   /* modifica y muestra el contenido de arreglo2 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "arreglo2[ %d ] = %d  ", i, arreglo2[ i ] += 5 );
   } /* fin de for */

} /* fin de la función iniciaArregloAutomatico */
```

```
Primera llamada a cada funcion:

Valores al entrar a iniciaArregloEstatico:
arreglo1[ 0 ] = 0 arreglo1[ 1 ] = 0 arreglo1[ 2 ] = 0
Valores al salir de iniciaArregloEstatico:
arreglo1[ 0 ] = 5 arreglo1[ 1 ] = 5  arreglo1[ 2 ] = 5
Valores al entrar a iniciaArregloAutomatico:
arreglo2[ 0 ] = 1  arreglo2[ 1 ] = 2  arreglo2[ 2 ] = 3
Valores al salir de iniciaArregloAutomatico:
arreglo2[ 0 ] = 6  arreglo2[ 1 ] = 7  arreglo2[ 2 ] = 8

Segunda llamada a cada funcion:

Valores al entrar a iniciaArregloEstatico:
arreglo1[ 0 ] = 5  arreglo1[ 1 ] = 5  arreglo1[ 2 ] = 5
Valores al salir de iniciaArregloEstatico:
arreglo1[ 0 ] = 10  arreglo1[ 1 ] = 10  arreglo1[ 2 ] = 10

Valores al entrar a iniciaArregloAutomatico:
arreglo2[ 0 ] = 1  arreglo2[ 1 ] = 2  arreglo2[ 2 ] = 3
Valores al salir de  iniciaArregloAutomatico:
arreglo2[ 0 ] = 6  arreglo2[ 1 ] = 7  arreglo2[ 2 ] = 8
```

**Figura 6.11** Si el programador no inicializa explícitamente los arreglos `static`, éstos se inicializan automáticamente en cero. (Parte 2 de 2.)

## 6.5 Cómo pasar arreglos a funciones

Para pasar un arreglo como argumento a una función, especifique el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo **tempCadaHora** se declara como

```
int tempCadaHora[ 24 ];
```

la llamada de función

```
modificaArreglo ( tempCadaHora, 24 )
```

pasa el arreglo **tempCadaHora** y su tamaño a la función **modificaArreglo**. A diferencia de los arreglos **char** que contienen cadenas, estos tipos de arreglos no tienen un terminador especial. Por esta razón, el tamaño del arreglo se pasa a la función, para que ésta pueda procesar el número apropiado de elementos.

C pasa automáticamente por referencia los arreglos a funciones; las funciones llamadas pueden modificar los valores del elemento en los arreglos originales de las funciones que las llaman. El nombre del arreglo es en realidad la dirección del primer elemento del arreglo. Debido a que se pasa la dirección inicial del arreglo, la función llamada conoce precisamente en dónde está almacenado el arreglo. Por lo tanto, cuando la función llamada modifica los elementos del arreglo en su cuerpo de función, modifica los elementos actuales del arreglo en sus posiciones originales de memoria.

La figura 6.12 demuestra que el nombre de un arreglo en realidad es la dirección del primer elemento del arreglo, imprimiendo **arreglo**, **&arreglo[ 0 ]** y **&arreglo** mediante el especificador de conversión **%p**; un especificador de conversión especial para la impresión de direcciones. El especificador de conversión **%p** normalmente despliega las direcciones como números hexadecimales. Los números hexadecimales (base 16) consisten en dígitos del 0 al 9 y las letras A a F (estas letras son los equivalentes hexadecimales de los números 10 a 15). Con frecuencia se utilizan como notación abreviada para valores enteros grandes. El apéndice E, Sistemas numéricos, proporciona una explicación profunda de las relaciones entre enteros binarios (base 2), octales (base 8), decimales (base 10; enteros estándar) y hexadecimales. La salida del programa muestra que tanto **arreglo** como **&arreglo[ 0 ]** tienen el mismo valor, a saber, 0065FDF0. La salida de este programa depende del sistema, pero las direcciones siempre son idénticas para una ejecución en particular de este programa en una computadora en particular.

```
/* Figura 6.12: fig06_12.c
   El nombre de un arreglo es lo mismo que &arreglo[ 0 ] */
#include <stdio.h>

  /* la función main comienza la ejecución del programa */
int main()
{
   char arreglo[ 5 ]; /* define un arreglo de 5 elementos */

   printf( “    arreglo = %p\n&arreglo[0] = %p\n”
      “   &arreglo = %p\n”,
      arreglo, &arreglo[ 0 ], &arreglo );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
    arreglo = 0065FDF0
&arreglo[0] = 0065FDF0
   &arreglo = 0065FDF0
```

**Figura 6.12** El nombre de un arreglo es el mismo que la dirección del primer elemento del arreglo.



### Tip de rendimiento 6.3

*Pasar arreglos por referencia tiene sentido por motivos de rendimiento. Si los arreglos se pasaran por valor, entonces una copia de cada elemento también pasaría. Esto implicaría que para pasar arreglos grandes y de manera frecuente, se requeriría demasiado tiempo y demasiado espacio de almacenamiento para las copias de los arreglos.*



### Observación de ingeniería de software 6.2

*Es posible pasar un arreglo por valor (mediante un simple truco que explicaremos en el capítulo 10).*

Aunque arreglos completos se pasan por referencia, los elementos individuales de un arreglo se pasan por valor, como se hace con variables sencillas. A tales conjuntos de datos individuales (como **`ints`**, **`float`**s y **`char`**s individuales) se les llama *escalares*. Para pasar un elemento de un arreglo a una función, utilice el nombre con subíndice del elemento, como un argumento en la llamada de función. En el capítulo 7, explicamos cómo pasar por referencia escalares (es decir, variables individuales y elementos de arreglos) a funciones.

Para que una función reciba un arreglo a través de una llamada de función, la lista de parámetros de la función debe especificar que se recibirá a un arreglo. Por ejemplo, el encabezado de función para la función **`modificaArreglo`** (que mencionamos anteriormente en esta sección) podría escribirse como

```
void modificaArreglo( int b[], int tamanio )
```

el cual indica que **`modificaArreglo`** espera recibir un arreglo de enteros en el parámetro **`b`** y el número de elementos del arreglo en el parámetro **`tamanio`**. No es necesario encerrar entre corchetes el tamaño del arreglo. Si éste se incluye, el compilador verifica si es mayor que cero para ignorarlo. Especificar un tamaño negativo genera un error de compilación. Debido a que los arreglos pasan automáticamente por referencia, cuando la función llamada utiliza el nombre de arreglo **`b`**, ésta hará referencia al arreglo de la función que llama (es decir, al arreglo **`tempCadaHora`** de la llamada anterior). En el capítulo 7, presentamos otras notaciones para indicar que un arreglo está siendo recibido por una función. Como veremos, estas notaciones se basan en la estrecha relación que existe entre los arreglos y los apuntadores en C.

La figura 6.13 demuestra la diferencia entre pasar un arreglo completo y pasar un elemento del arreglo. El programa primero imprime los cinco elementos del entero arreglo a (líneas 20 a 22). Después, **`a`** y su tamaño pasan a la función **`modificaArreglo`** (línea 27), donde cada uno de los elementos del arreglo **`a`** se multiplica por 2 (líneas 56 y 57). Posteriormente, **`a`** se vuelve a imprimir en **`main`** (líneas 32 a 34). Como muestra la salida, los elementos del arreglo **`a`** en realidad son modificados por **`modificaArreglo`**. Ahora el programa imprime el valor de **`a[ 3 ]`** (línea 38) y lo pasa a la función **`modificaElemento`** (línea 40). La función **`modificaElemento`** multiplica su argumento por 2 (línea 67) e imprime el nuevo valor. Observe que cuando **`a[ 3 ]`** se vuelve a imprimir en **`main`** (línea 43), no se ha modificado, ya que los elementos individuales de un arreglo se pasan por valor.

```
1  /* Figura 6.13: fig06_13.c
2     Paso de arreglos y de elementos de un arreglo a funciones */
3  #include <stdio.h>
4  #define TAMANIO 5
5
6  /* prototipos de las funciones */
7  void modificaArreglo( int b[], int tamanio );
8  void modificaElemento( int e );
9
10 /* la función main comienza la ejecución del programa */
11 int main()
12 {
13    int a[ TAMANIO ] = { 0, 1, 2, 3, 4 }; /* inicializa a */
14    int i; /* contador */
15
```

**Figura 6.13** Paso de arreglos y de elementos individuales a funciones. (Parte 1 de 3.)

```
   printf( “Efectos de pasar arreglos completos por referencia:\n\nlos “
           “valores del arreglo original son:\n” );

   /* muestra el arreglo original */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%3d”, a[ i ] );
   } /* fin de for */

   printf( “\n” );

   /* pasa el arreglo a modificaArreglo por referencia */
   modificaArreglo( a, TAMANIO );

   printf( “Los valores del arreglo modificado son:\n” );

   /* muestra el arreglo modificado */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%3d”, a[ i ] );
   } /* fin de for */

   /* muestra el valor de a[ 3 ] */
   printf( “\n\n\nEfectos de pasar un elemento del arreglo “
            “por valor:\n\nEl valor de a[3] es %d\n”, a[ 3 ] );

   modificaElemento( a[ 3 ] ); /* pasa el elemento a[ 3 ] del arreglo por
   valor */

   /* muestra el valor a[ 3 ] */
   printf( “El valor de a[ 3 ] es %d\n”, a[ 3 ] );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* en la función modificaArreglo, “b” apunta al arreglo original “a”
   en memoria */
void modificaArreglo( int b[], int tamanio )
{
   int j; /* contador */

   /* multiplica cada elemento del arreglo por 2 */
   for ( j = 0; j < tamanio; j++ ) {
      b[ j ] *= 2;
   } /* fin de for */

} /* fin de la función modificaArreglo */

/* en la función modificaElemento, “e” es una copia local del elemento a[ 3 ]
   del arreglo se pasó desde main */
void modificaElemento( int e )
{
   /* multiplica el parámetro por 2 */
   printf( “El valor en modificaElemento es %d\n”, e *= 2 );
} /* fin de la función modificaElemento */
```

**Figura 6.13** Paso de arreglos y de elementos individuales a funciones. (Parte 2 de 3.)

```
Efectos de pasar arreglos completos por referencia:

los valores del arreglo original son:
  0  1  2  3  4
Los valores del arreglo modificado son:
  0  2  4  6  8


Efectos de pasar un elemento del arreglo por valor:

El valor de a[3] es 6
El valor en modificaElemento es 12
El valor de a[ 3 ] es 6
```

**Figura 6.13** Paso de arreglos y de elementos individuales a funciones. (Parte 3 de 3.)

Pueden existir situaciones en sus programas en las que no se debe permitir que una función modifique los elementos de un arreglo. Debido a que los arreglos siempre se pasan por referencia, la modificación de valores de arreglos es difícil de controlar. C proporciona el calificador de tipo ***const*** para prevenir que una función modifique los valores de un arreglo. Cuando un parámetro de un arreglo es precedido por el calificador **const**, los elementos del arreglo se vuelven constantes en el cuerpo de la función, y cualquier intento de modificar un elemento del arreglo en el cuerpo de la función da como resultado un error en tiempo de compilación. Esto permite al programador corregir un programa para que no intente modificar los elementos de un arreglo.

La figura 6.14 muestra el calificador **const**. La función **intentaModifElArreglo** (línea 22) se declara con el parámetro **const int b[]**, el cual especifica que el arreglo **b** es constante y no puede modificarse. La salida muestra el mensaje de error que produce el compilador; los errores pueden ser diferentes en su sistema. Cada uno de los tres intentos que hace la función de modificar los elementos del arreglo, da como resultado el error del compilador "**l-value specifies a const object**". En el capítulo 7 explicamos nuevamente el calificador **const**.

### Observación de ingeniería de software 6.3

*El calificador de tipo **const** puede aplicarse a un parámetro de arreglo en la declaración de una función, para prevenir que el arreglo original sea modificado en el cuerpo de la función. Éste es otro ejemplo del principio del menor privilegio. A las funciones no se les debe dar la capacidad de modificar un arreglo, a menos que sea absolutamente necesario.*

```
1  /* Figura 6.14: fig06_14.c
2     Demostración del calificador de tipo const con arreglos */
3  #include <stdio.h>
4
5  void intentaModifElArreglo( const int b[] ); /* prototipo de la función */
6
7  /* la función main comienza la ejecución del programa */
8  int main()
9  {
10    int a[] = { 10, 20, 30 }; /* inicializa a */
11
12    intentaModifElArreglo( a );
13
14    printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15
16    return 0; /* indica terminación exitosa */
```

**Figura 6.14** Calificador de tipo **const**. (Parte 1 de 2.)

```

} /* fin de main */

/* en la función intentaModifElArreglo, el arreglo b es const, por lo tanto
   no puede ser
   utilizado para modificar el arreglo original a en main. */
void intentaModifElArreglo( const int b[] )
{
   b[ 0 ] /= 2;    /* error */
   b[ 1 ] /= 2;    /* error */
   b[ 2 ] /= 2;    /* error */
} /* fin de la función intentaModifElArreglo */
```

```
Compiling...
fig06_14.c
\\DEITEL\CH06\fig06_14.c(24) : error C2166: l-value specifies const object
\\DEITEL\CH06\fig06_14.c(25) : error C2166: l-value specifies const object
\\DEITEL\CH06\fig06_14.c(26) : error C2166: l-value specifies const object
```

**Figura 6.14** Calificador de tipo **`const`**. (Parte 2 de 2.)

## 6.6 Ordenamiento de arreglos

El *ordenamiento* de datos (es decir, colocar los datos en un orden particular, ya sea ascendente o descendente) es una de las aplicaciones de cómputo más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que puede preparar los estados individuales del banco al final de cada mes. Las empresas de telefonía ordenan sus listas de cuentas por apellido y, dentro de este ordenamiento, hacen otro por nombre para facilitar la búsqueda de números telefónicos. Virtualmente todas las empresas deben ordenar algún tipo de dato y, en muchos casos, cantidades masivas de éstos. El ordenamiento de datos es un problema intrigante que ha dado pie a algunas de las acciones de investigación más intensas en el campo de las ciencias de la computación. En este capítulo explicamos el método de ordenamiento más sencillo. En los ejercicios y en el capítulo 12, investigamos métodos más complejos que logran un mejor rendimiento.



**Tip de rendimiento 6.4**

*Algunas veces, los algoritmos más sencillos tienen un rendimiento muy pobre. Su virtud radica en que son fáciles de escribir, probar y depurar. Sin embargo, los algoritmos más complejos son necesarios para lograr un máximo rendimiento.*

La figura 6.15 ordena de manera ascendente los valores que corresponden a los elementos del arreglo a (línea 10). La técnica que utilizamos es conocida como *ordenamiento burbuja* o *método de hundimiento*, ya que los valores más pequeños "flotan" gradualmente hacia arriba, hacia el encabezado del arreglo, como burbujas de aire hacia la superficie del agua, mientras que los valores más grandes se hunden en el fondo del arreglo. La técnica es para realizar varias pasadas a través del arreglo. En cada pasada, se comparan pares sucesivos de elementos. Si el par está en orden ascendente (o si los valores son idénticos), dejamos los valores como están. Si el par se encuentra en orden decreciente, sus valores se intercambian en el arreglo.

```
1 /* Figura 6.15: fig06_15.c
2    Este programa ordena los valores de un arreglo en orden ascendente */
3 #include <stdio.h>
4 #define TAMANIO 10
5
```

**Figura 6.15** Ordenamiento de un arreglo mediante el ordenamiento burbuja. (Parte 1 de 2.)

```
/* la función main comienza la ejecución del programa */
int main()
{
   /* inicializa a */
   int a[ TAMANIO ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int pasadas;  /* contador de pasadas */
   int i;        /* contador de comparaciones */
   int almacena; /* ubicación temporal utilizada para el intercambio de
                    elementos */

   printf( “Elementos de datos en el orden original\n” );

   /* muestra el arreglo original */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%4d”, a[ i ] );
   } /* fin de for */

   /* ordenamiento burbuja */
   /* ciclo para controlar el número de pasos */
   for ( pasadas = 1; pasadas < TAMANIO; pasadas++ ) {

      /* ciclo para controlar el número de comparaciones por pasada */
      for ( i = 0; i < TAMANIO - 1; i++ ) {

         /* compara los elementos adyacentes y los intercambia si el primer
         elemento es mayor que el segundo */
         if ( a[ i ] > a[ i + 1 ] ) {
            almacena = a[ i ];
            a[ i ] = a[ i + 1 ];
            a[ i + 1 ] = almacena;
         } /* fin de if */

      } /* fin del for interno */

   } /* fin del for externo */

   printf( “\nElementos de datos en orden ascendente\n” );

   /* muestra el arreglo ordenado */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%4d”, a[ i ] );
   } /* fin de for */

   printf( “\n” );

   return 0; /* indica terminación exitosa */

```

```
Elementos de datos en el orden original
   2   6   4   8  10  12  89  68  45  37
Elementos de datos en orden ascendente
   2   4   6   8  10  12  37  45  68  89
```

**Figura 6.15** Ordenamiento de un arreglo mediante el ordenamiento burbuja. (Parte 2 de 2.)

Primero, el programa compara **a[ 0 ]** con **a[ 1 ]**, después **a[ 1 ]** con **a[ 2 ]**, luego **a[ 2 ]** con **a[ 3 ]**, y así sucesivamente hasta que completa la pasada, comparando **a[ 8 ]** con **a[ 9 ]**. Observe que aunque hay 10 elementos, solamente se realizan 9 comparaciones. Debido a la manera en que se realizan las comparaciones sucesivas, un valor grande puede moverse muchas posiciones hacia abajo en una sola pasada, pero un valor pequeño puede moverse sólo una posición hacia arriba. En la primera pasada, se garantiza que el valor más grande se hunde hasta el fondo del arreglo, **a[ 9 ]**. En la segunda pasada, el segundo valor más grande se hunde hasta **a[ 8 ]**. En la novena pasada, el noveno valor más grande se hunde hasta **a[ 1 ]**. Esto deja al valor más pequeño en **a[ 0 ]**, por lo que sólo se necesitan nueve pasadas para ordenar el arreglo, aunque éste tenga 10 elementos.

El ordenamiento se realiza por medio del ciclo anidado **for** (líneas 24 a 39). Si es necesario realizar un intercambio, éste se lleva a cabo por medio de las tres asignaciones siguientes

```
almacena = a[ i ];
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = almacena;
```

en donde la variable adicional **almacena**, guarda temporalmente uno de los dos valores a intercambiar. El intercambio no puede llevarse a cabo únicamente con las asignaciones

```
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = a[ i ];
```

Si, por ejemplo, **a[ i ]** es **7** y **a[ i + 1 ]** es **5**, después de la primera asignación ambos valores serán **5**, y el valor **7** se perderá. De aquí la necesidad de la variable adicional **almacena**.

La principal virtud del ordenamiento burbuja es que es fácil de programar, sin embargo, es lento. Esto se hace evidente cuando se ordenan arreglos grandes. En los ejercicios desarrollaremos versiones más eficientes del ordenamiento burbuja, e investigaremos algunos métodos más eficientes que éste. En cursos más avanzados se analizan con detalle el ordenamiento y la búsqueda.

## 6.7 Ejemplo práctico: Cálculo de la media, la mediana y la moda a través de arreglos

Ahora consideraremos un ejemplo más grande. Por lo general, las computadoras se utilizan para compilar y analizar los resultados de *encuestas* y estudios de opinión. La figura 6.16 utiliza el arreglo **respuesta**, el cual inicializa con 99 respuestas de una encuesta. Cada respuesta es un número del 1 al 9. El programa calcula la media, la mediana y la moda de los 99 valores.

```
/* Figura 6.16: fig06_16.c
   Este programa introduce el tema del análisis de datos.
   Calcula la media, la mediana y la moda de los datos */
#include <stdio.h>
#define TAMANIO 99

/* prototipos de las funciones */
void media( const int resp[] );
void mediana( int resp[] );
void moda( int frec[], const int resp[] ) ;
void ordenamBurbuja( int a[] );
void imprimeArreglo( const int a[] );

/* la función main comienza la ejecución del programa */
int main()
{
   int frecuencia[ 10 ] = { 0 }; /* inicializa el arreglo frecuencia */

```

**Figura 6.16** Programa para el análisis de los datos de una encuesta. (Parte 1 de 4.)

```
    /* inicializa el arreglo respuestas */
    int respuesta[ TAMANIO ] =
       { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
         4, 5, 6, 1, 6, 5, 7, 8, 7 };

    /* procesa las respuestas */
    media( respuesta );
    mediana( respuesta );
    moda( frecuencia, respuesta );

    return 0; /* indica terminación exitosa */

} /* fin de main */

/* calcula el promedio de todos los valores de las respuestas */
void media( const int resp[] )
{
    int j; /* contador del total de elementos del arreglo */
    int total = 0; /* variable para mantener la suma de los elementos del
                      arreglo */

    printf( “%s\n%s\n%s\n”, “********”, “  Media”, “********” );

    /* total de los valores de las respuestas */
    for ( j = 0; j < TAMANIO; j++ ) {
       total += resp[ j ];
    } /* fin de for */

    printf( “La media es el valor promedio de los datos.\n”
            “La media es igual al total de\n”
            “todos los elementos de datos divididos por\n”
            “el numero de elementos de datos ( %d ). La media\n”
            “en esta ejecucion es: %d / %d = %.4f\n\n”,
            TAMANIO, total, TAMANIO, (  double  ) total / TAMANIO );
} /* fin de la función media */

/* ordena el arreglo y determina el valor de la mediana */
void mediana( int resp[] )
{
    printf( “\n%s\n%s\n%s\n%s”,
            “********”, “ Mediana”, “********”,
            “El arreglo de respuestas desordenado es” );

    imprimeArreglo( resp ); /* muestra el arreglo desordenado */

    ordenamBurbuja( resp ); /* ordena el arreglo */

```

**Figura 6.16** Programa para el análisis de los datos de una encuesta. (Parte 2 de 4.)

```
   printf( “\n\nEl arreglo ordenado es “ );
   imprimeArreglo( resp ); /* muestra el arreglo ordenado */

   /* muestra la mediana */
   printf( “\n\nLa mediana es el elemento %d del\n”
           “arreglo ordenado de elementos %d.\n”
           “Para esta ejecucion la mediana es %d\n\n”,
           TAMANIO / 2, TAMANIO, resp[ TAMANIO / 2 ] );
} /* fin de la función mediana */

/* determina las respuestas más frecuentes */
void moda( int frec[], const int resp[] )
{
   int rango; /* contador para acceder a los elementos de 1 a 9 del arreglo
                frec */
   int j; /* contador para sumar los elementos de 0 a 98 des arreglo
             respuesta */
   int h; /* contador para desplegar histogramas de los elementos en el
             arreglo frec */
   int masGrande = 0; /* representa la frecuencia más grande */
   int valorModa = 0; /* representa la respuesta más frecuente */

   printf( “\n%s\n%s\n%s\n”,
           “********”, “  Moda”, “********” );

   /* inicializa las frecuencias a 0 */
   for ( rango = 1; rango <= 9; rango++ ) {
      frec[ rango ] = 0;
   } /* fin de for */

   /* suma las frecuencias */
   for ( j = 0; j < TAMANIO; j++ ) {
      ++frec[ resp[ j ] ];
   } /* fin de for */

   /* muestra los encabezados de las columnas */
   printf( “%s%11s%19s\n\n%54s\n%54s\n\n”,
           “Respuesta”, “Frecuencia”, “Histograma”,
           “1    1    2    2”, “5    0    5    0    5” );

   /* muestra los resultados */
   for ( rango = 1; rango <= 9; rango++ ) {
      printf( “%8d%11d          “, rango, frec[ rango ] );

      /* sigue la pista del valor de la moda y del valor de la frecuencia
         más grande */
      if ( frec[ rango ] > masGrande ) {
         masGrande = frec[ rango ];
         valorModa = rango;
      } /* fin de if */

      /* muestra los histogramas de barras que representan el valor de la
         frecuencia */
      for ( h = 1; h <= frec[ rango ]; h++ ) {
         printf( “*” );
```

**Figura 6.16** Programa para el análisis de los datos de una encuesta. (Parte 3 de 4.)

```
      } /* fin del for interno */

      printf( "\n" ); /* nueva línea de salida */
   } /* fin del for externo */

   /* despliega el valor de la moda */
   printf( "La moda es el valor mas frecuente.\n"
           "Para esta ejecucion la moda es %d el cual ocurrio"
           " %d veces.\n", valorModa, masGrande );
} /* fin de la función moda */

/* función que ordena un arreglo mediante el algoritmo del método de la
   burbuja algorithm */
void ordenamBurbuja( int a[] )
{
   int pasada;   /* contador de pasadas */
   int j;        /* contador de comparaciones */
   int almacena; /* ubicación temporal utilizada para intercambiar los
                    elementos */

   /* ciclo para controlar el número de pasadas */
   for ( pasada = 1; pasada < TAMANIO; pasada++ ) {

      /* ciclo para controlar el número de comparaciones por pasada */
      for ( j = 0; j < TAMANIO - 1; j++ ) {

         /* intercambia los elementos si no se encuentran en orden */
         if ( a[ j ] > a[ j + 1 ] ) {
            almacena = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = almacena;
         } /* fin de if */

      } /* fin del for interno */

   } /* fin del for externo */

} /* fin de ordenamBurbuja */

/* muestra el contenido del arreglo (20 valores por línea) */
void imprimeArreglo( const int a[] )
{
   int j; /* contador */

   /* muestra el contenido del arreglo */
   for ( j = 0; j < TAMANIO; j++ ) {

      if ( j % 20 == 0 ) { /* comienza una nueva línea cada 20 valores */
         printf( "\n" );
      } /* fin de if */

      printf( "%2d", a[ j ] );
   } /* fin de for */

} /* fin de la función imprimeArreglo */
```

**Figura 6.16** Programa para el análisis de los datos de una encuesta. (Parte 4 de 4.)

La media es el promedio aritmético de los 99 valores. La función **media** (línea 42) calcula el promedio sumando el valor de los 99 elementos y dividiendo el resultado entre 99.

La mediana es el "valor medio". La función **mediana** (línea 63) determina la mediana a través de la llamada a la función **ordenamBurbuja** (declarada en la línea 135), para ordenar de manera ascendente el arreglo de respuestas y obtener el elemento central, **resp[ TAMANIO / 2 ]**, del arreglo ordenado. Observe que cuando hay un número par de elementos, la mediana debe calcularse como el promedio de los dos elementos centrales. La función **mediana** actualmente no proporciona esta capacidad. A la función **imprimeArreglo** (línea 161) se le llama para que despliegue el arreglo **respuesta**.

La moda es el valor más frecuente entre las 99 respuestas. La función **moda** (línea 84) determina la moda contando el número de respuestas de cada tipo y posteriormente seleccionando el valor con más ocurrencias. Esta versión de la función **moda** no maneja un vínculo (vea el ejercicio 6.14). La función **moda** también produce un histograma para ayudar a determinar la moda de manera gráfica. La figura 6.17 contiene un ejemplo de la ejecución de este programa. Este ejemplo incluye la mayoría de las manipulaciones comunes que generalmente se necesitan en problemas relacionados con arreglos, incluso el paso de arreglos a funciones.

## 6.8 Búsqueda en arreglos

Con frecuencia, un programador se verá trabajando con grandes volúmenes de datos almacenados en arreglos. Podría ser necesario determinar si un arreglo contiene un valor que coincide con cierto *valor clave*. Al proceso

```
********
  Media
********
La media es el valor promedio de los datos.
La media es igual al total de
todos los elementos de datos divididos por
el numero de elementos de datos ( 99 ). La media
en esta ejecucion es: 681 / 99 = 6.8788

********
 Mediana
********
El arreglo de respuestas desordenado es
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

El arreglo ordenado es
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

La mediana es el elemento 49 del
arreglo ordenado de 99 elementos.
Para esta ejecucion la mediana es 7

********
  Moda
********
```

**Figura 6.17** Ejemplo de la ejecución del programa para analizar los datos de una encuesta. (Parte 1 de 2.)

```
Respuesta Frecuencia         Histograma

                                      1    1    2    2
                                 5    0    5    0    5

       1          1           *
       2          3           ***
       3          4           ****
       4          5           *****
       5          8           ********
       6          9           *********
       7         23           ***********************
       8         27           ***************************
       9         19           *******************
La moda es el valor mas frecuente.
Para esta ejecucion la moda es 8 el cual ocurrio 27 veces.
```

**Figura 6.17** Ejemplo de la ejecución del programa para analizar los datos de una encuesta. (Parte 2 de 2.)

de encontrar un elemento en particular de un arreglo se le conoce como *búsqueda*. En esta sección, explicamos dos técnicas de búsqueda: la técnica simple de *búsqueda lineal*, y una más eficiente, pero más compleja, la técnica de *búsqueda binaria*. Los ejercicios 6.34 y 6.35 al final de este capítulo le pedirán que implemente versiones recursivas de la búsqueda lineal y de la búsqueda binaria.

***Búsqueda en un arreglo mediante la búsqueda lineal***

La búsqueda lineal (figura 6.18) compara cada elemento de un arreglo con la *clave de búsqueda*. Debido a que el arreglo no se encuentra en un orden particular, la probabilidad de que el valor se encuentre en el primer elemento o en el último, es la misma. Por lo tanto, en promedio, el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del arreglo.

```
/* Figura 6.18: fig06_18.c
   Búsqueda lineal en un arreglo */
#include <stdio.h>
#define TAMANIO 100

/* prototipo de la función */
int busquedaLineal( const int arreglo[], int llave, int tamanio );

/* la función main comienza la ejecución del programa */
int main()
{
   int a[ TAMANIO ]; /* crea el arreglo a */
   int x; /* contador para inicializar los elementos de 0 a 99 del arreglo a */
   int llaveBusqueda; /* valor para localizar en el arreglo a */
   int elemento; /* variable para almacenar la ubicación de llaveBusqueda o -1 */

   /* crea los datos */
   for ( x = 0; x < TAMANIO; x++ ) {
      a[ x ] = 2 * x;
   } /* fin de for */

   printf( "Introduzca la llave de busqueda entera:\n" );
```

**Figura 6.18** Búsqueda lineal en un arreglo. (Parte 1 de 2.)

```
   scanf( "%d", &llaveBusqueda );

   /* intenta localizar llaveBusqueda en el arreglo a */
   elemento = busquedaLineal( a, llaveBusqueda, TAMANIO );

   /* despliega los resultados */
   if ( elemento != -1 ) {
      printf( "Encontre el valor en el elemento %d\n", elemento );
   } /* fin de if */
   else {
      printf( "Valor no encontrado\n" );
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* compara la llave con cada elemento del arreglo hasta que localiza el elemento
   o hasta que alcanza el final del arreglo; devuelve el subíndice del elemento
   si lo encontró o -1 si no lo encontró */
int busquedaLineal( const int arreglo[], int llave, int tamanio )
{
   int n; /* contador */

   /* ciclo a través del arreglo */
   for ( n = 0; n < tamanio; ++n ) {

      if ( arreglo[ n ] == llave ) {
         return n; /* devuelve la ubicación de la llave */
      } /* fin de if */

   } /* fin de for */

   return -1; /* llave no encontrada */

} /* fin de la función busquedaLineal */
```

```
Introduzca la llave de busqueda entera:
36
Encuentre el valor en el elemento 18
```

```
Introduzca la llave de busqueda entera:
37
Valor no encontrado
```

**Figura 6.18** Búsqueda lineal en un arreglo. (Parte 2 de 2.)

### *Búsqueda en un arreglo mediante la búsqueda binaria*

El método de búsqueda lineal trabaja bien para arreglos pequeños o para arreglos desordenados. Sin embargo, para arreglos grandes, la búsqueda lineal es ineficiente. Si el arreglo se encuentra ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria.

Después de cada comparación, el algoritmo de la búsqueda binaria elimina la mitad de los elementos del arreglo ordenado en el que se busca. El algoritmo localiza el elemento central de un arreglo y lo compara con la clave de búsqueda. Si son iguales, entonces localizó la clave de búsqueda, y devuelve el subíndice del elemento del arreglo. De lo contrario, el problema se reduce a buscar en una mitad del arreglo. Si la clave de búsqueda es menor que el elemento central del arreglo, la búsqueda se realiza en la primera mitad de éste; de lo contrario, la búsqueda se realiza en la segunda mitad. Si la clave de búsqueda no se encuentra en el subarreglo especificado (parte del arreglo original), el algoritmo se repite en un cuarto del arreglo original. La búsqueda continúa hasta que la clave de búsqueda es igual al elemento central de un subarreglo, o hasta que el subarreglo consista en un elemento que no sea igual a la clave de búsqueda (es decir, no se encontró la clave de búsqueda).

En el peor de los casos, para realizar una búsqueda en un arreglo de 1023 elementos por medio de la búsqueda binaria, sólo se necesitarán 10 comparaciones. Dividir de manera repetida 1024 entre 2 arroja los valores 512, 256, 128, 64, 32, 16, 8, 4, 2 y 1. El número 1024 ($2^{10}$) se divide entre 2 sólo diez veces para obtener el valor de 1. Dividir entre 2 es equivalente a realizar una comparación con el algoritmo de la búsqueda binaria. Un arreglo de 1048576 ($2^{20}$) toma un máximo de 20 comparaciones para encontrar la clave de búsqueda. Un arreglo de mil millones de elementos toma un máximo de 30 comparaciones para encontrar la clave de búsqueda. Éste es un aumento tremendo de rendimiento con respecto a la búsqueda lineal, la cual requiere comparar la clave de búsqueda con un promedio aproximado de la mitad de los elementos de un arreglo. Para un arreglo de mil millones de elementos, ¡ésta es una diferencia promedio de 500 millones de comparaciones y un máximo de 30 comparaciones! El número máximo de comparaciones para cualquier arreglo se puede determinar buscando la primera potencia de 2 mayor que el número de elementos en el arreglo.

La figura 6.19 presenta la versión iterativa de la función **busquedaBinaria** (líneas 45 a 77). La función recibe cuatro argumentos: un arreglo entero **b** en el que se realizará la búsqueda, una **claveDeBusqueda** entera, el subíndice **bajo** del arreglo y el subíndice **alto** del arreglo (éstos se definen en la parte del arreglo en el que se va a buscar). Si la clave de búsqueda no coincide con el elemento central de un subarreglo, el subíndice **bajo** o el subíndice **alto** se modifican para que se pueda buscar en un subarreglo más pequeño. Si la clave de búsqueda es menor que el elemento central, el subíndice **alto** se establece en **central-1**, y la búsqueda continúa sobre los elementos desde **bajo** hasta **central-1**. Si la clave de búsqueda es mayor que el elemento central, el subíndice **bajo** se establece en **central + 1**, y la búsqueda continúa sobre los elementos desde **bajo** hasta **central + 1**. El programa utiliza un arreglo de 15 elementos. La primera potencia de 2 que resulta mayor que el número de elementos de este arreglo es 16 ($2^4$), por lo que sólo se necesitan 4 comparaciones para encontrar la clave de búsqueda. El programa utiliza una función **despliegaEncabezado** (líneas 80 a 99) para desplegar los subíndices del arreglo, y la función **despliegaLinea** (líneas 103 a 124) para desplegar cada subarreglo durante el proceso de búsqueda binaria. El elemento central de cada subarreglo se marca con un asterisco (*****) para indicar el elemento con el que se compara la clave de búsqueda.

```
1  /* Figura 6.19: fig06_19.c
2     Búsqueda binaria dentro de un arreglo */
3  #include <stdio.h>
4  #define TAMANIO 15
5
6  /* prototipos de las funciones */
7  int busquedaBinaria( const int b[], int claveDeBusqueda, int bajo, int alto );
8  void despliegaEncabezado( void );
9  void despliegaLinea( const int b[], int bajo, int medio, int alto );
10
11 /* la función main comienza la ejecución del programa */
12 int main()
13 {
14    int a[ TAMANIO ]; /* crea el arreglo a */
15    int i; /* contador para inicializar los elementos de 0 a 14 del arreglo a */
```

**Figura 6.19** Búsqueda lineal en un arreglo ordenado. (Parte 1 de 4.)

```
    int llave; /* valor a localizar en el arreglo a */
    int resultado; /* variable para almacenar la ubicación de la llave o -1 */

    /* crea los datos */
    for ( i = 0; i < TAMANIO; i++ ) {
       a[ i ] = 2 * i;
    } /* fin de for */

    printf( “Introduzca un numero entre 0 y 28: “ );
    scanf( “%d”, &llave );

    despliegaEncabezado();

    /* busca la llave en el arreglo a */
    resultado = busquedaBinaria( a, llave, 0, TAMANIO - 1 );

    /* despliega los resultados */
    if ( resultado != -1 ) {
       printf( “\n%d se encuentra en el elemento %d del arreglo\n”, llave,
       resultado );
    } /* fin de if */
    else {
       printf( “\n%d no se encuentra\n”, llave );
    } /* fin de else */

    return 0; /* indica terminación exitosa */

} /* fin de main */

/* función para realizar la búsqueda binaria en un arreglo */
int busquedaBinaria( const int b[], int claveDeBusqueda, int bajo, int alto )
{
   int central; /* variable para mantener el elemento central del arreglo */

    /* realiza el ciclo hasta que el subínice bajo es mayor que el subíndice
       alto */
    while ( bajo <= alto ) {

       /* determina el elemento central del subarreglo en el que se busca */
       central = ( bajo + alto ) / 2;

       /* despliega el subarreglo utilizado en este ciclo */
       despliegaLinea( b, bajo, central, alto );

       /* si claveDeBusqueda coincide con el elemento central, devuelve
          central */
       if ( claveDeBusqueda == b[ central ] ) {
          return central;
       } /* fin de if */

       /* si claveDeBusqueda es menor que el elemento central, establece el
          nuevo valor de alto */
       else if ( claveDeBusqueda < b[ central ] ) {
          alto = central - 1; /* busca en la mitad inferior del arreglo */
       } /* fin de else if */
```

**Figura 6.19** Búsqueda lineal en un arreglo ordenado. (Parte 2 de 4.)

```
      /* si claveDeBusqueda es mayor que el elemento central, establece el
         nuevo valor para bajo */
      else {
         bajo = central + 1; /* busca en la mitad superior del arreglo */
      } /* fin de else */

   } /* fin de while */

   return -1;   /* no se encontró claveDeBusqueda */

} /* fin de la función busquedaBinaria */

/* Imprime un encabezado para la salida */
void despliegaEncabezado( void )
{
   int i; /* contador */

   printf( “\nSubindices:\n” );

   /* muestra el encabezado de la columna */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%3d “, i );
   } /* fin de for */

   printf( “\n” ); /* comienza la nueva línea de salida */

   /* muestra una línea de caracteres  - */
   for ( i = 1; i <= 4 * TAMANIO; i++ ) {
      printf( “-” );
   } /* fin de for */

   printf( “\n” ); /* inicia una nueva línea de salida */
} /* fin de la función despliegaEncabezado */

/* Imprime una línea de salida que muestra la parte actual
   del arreglo que se está procesando. */
void despliegaLinea( const int b[], int baj, int cen, int alt )
{
   int i; /* contador para la iteración a través del arreglo b */

   /* ciclo a través del arreglo completo */
   for ( i = 0; i < TAMANIO; i++ ) {

      /* despliega espacios si se encuentra fuera del rango actual del
         subarreglo */
      if ( i < baj || i > alt ) {
         printf( “    “ );
      } /* fin de if */
      else if ( i == cen ) { /* despliega el elemento central */
         printf( “%3d*”, b[ i ] ); /* marca el valor central */
      } /* fin de else if */
      else { /* despliega otros elementos en el subarreglo */
         printf( “%3d “, b[ i ] );
      } /* fin de else */
```

**Figura 6.19** Búsqueda lineal en un arreglo ordenado. (Parte 3 de 4.)

```

    } /* fin de for */

    printf( "\n" ); /* inicia la nueva línea de salida */
} /* fin de la función despliegaLinea */
```

```
Introduzca un numero entre 0 y 28: 25

Subindices:
  0   1   2   3   4   5   6   7    8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 no se encuentra
```

```
Introduzca un numero entre 0 y 28: 8

Subindices:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 se encuentra en el elemento 4 del arreglo
```

```
Introduzca un numero entre 0 y 28: 6

Subindices:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 se encuentra en el elemento 3 del arreglo
```

**Figura 6.19** Búsqueda lineal en un arreglo ordenado. (Parte 4 de 4.)

## 6.9 Arreglos con múltiples subíndices

Los arreglos en C pueden tener múltiples subíndices. Un uso común de los arreglos con múltiples subíndices es la representación de *tablas* de valores que constan de información organizada en *filas* y *columnas*. Para identificar un elemento particular de una tabla, debemos especificar dos subíndices: el primero (por convención) identifica la fila del elemento, y el segundo (por convención) identifica la columna del elemento. A las tablas o arreglos que requieren dos subíndices para identificar un elemento particular se les conoce como *arreglos con dos subíndices*. Observe que los arreglos con múltiples subíndices pueden tener más de dos subíndices.

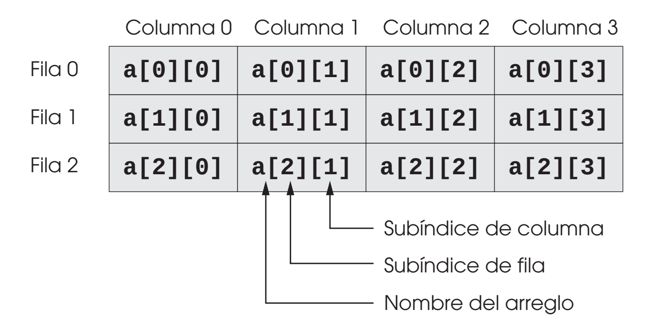



**Figura 6.20** Arreglo con dos subíndices con tres filas y cuatro columnas.

La figura 6.20 presenta el arreglo con dos subíndices, **`a`**. El arreglo contiene tres filas y cuatro columnas, por lo que se dice que es un arreglo de 3 por 4. En general, un arreglo con *m* filas y *n* columnas se conoce como *arreglo de m por n*.

Cada elemento del arreglo **`a`**, correspondiente a la figura 6.20, está identificado por un elemento nombre de la forma **`a[ i ][ j ]`**; **`a`** es el nombre del arreglo, e **`i`** y **`j`** son los subíndices que identifican de manera única a cada elemento de **`a`**. Observe que los nombres de los elementos de la primera fila tienen un primer subíndice de **`0`**; los nombres de los elementos de la cuarta columna tienen un segundo subíndice de 3.

### Error común de programación 6.9

*Hacer referencia a un elemento de un arreglo con dos subíndices de la forma* **`a[ x, y ]`**, *en lugar de hacerlo de la forma* **`a[ x ][ y ]`**.

Un arreglo con múltiples subíndices puede inicializarse en su declaración, de manera muy similar a un arreglo con un solo subíndice. Por ejemplo, un arreglo con dos subíndices **`int b[ 2 ][ 2 ]`** podría declararse e inicializarse con

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Los valores se agrupan por fila entre llaves. Los valores del primer conjunto de llaves inicializan la fila **`0`**, y los valores del segundo conjunto de llaves inicializan la fila **`1`**. Entonces, los valores **`1`** y **`2`** inicializan los elementos **`int b[ 0 ][ 0 ]`** y **`int b[ 0 ][ 1 ]`**, respectivamente, y los valores **`3`** y **`4`** inicializan los elemento **`int b[ 1 ][ 0 ]`** y **`int b[ 1 ][ 1 ]`**, respectivamente. Si no hay suficientes inicializadores para una fila dada, el resto de los elementos de esa fila se inicializan en **`0`**. Por lo tanto,

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

inicializaría **`b[ 0 ][ 0 ]`** en **`1`**, **`b[ 0 ][ 1 ]`** en **`0`**, **`b[ 1 ][ 0 ]`** en **`3`** y **`b[ 1 ][ 1 ]`** en **`4`**. La figura 6.21 muestra la declaración y la inicialización de arreglos con dos subíndices.

```
/* Figura 6.21: fig06_21.c
   Inicialización de arreglos multidimensionales */
#include <stdio.h>

void despliegaArreglo ( const int a[][ 3 ] ); /* prototipo de la función */

/* la función main comienza la ejecución del programa */
int main()
{
   /* inicializa arreglo1, arreglo2, arreglo3 */
   int arreglo1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int arreglo2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
```

**Figura 6.21** Inicialización de arreglos multidimensionales. (Parte 1 de 2.)

```
13    int arreglo3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    printf( “Los valores en el arreglo1 por línea son:\n” );
16    despliegaArreglo ( arreglo1 );
17
18    printf( “Los valores en el arreglo2 por línea son:\n” );
19    despliegaArreglo ( arreglo2 );
20
21    printf( “Los valores en el arreglo3 por línea son:\n” );
22    despliegaArreglo ( arreglo3 );
23
24    return 0; /* indica terminación exitosa */
25
26 } /* fin de main */
27
28 /* función para mostrar un arreglo con dos filas y tres columnas */
29 void despliegaArreglo ( const int a[][ 3 ] )
30 {
31    int i; /* contador de filas */
32    int j; /* contador de clumnas */
33
34    /* ciclo a través de las filas */
35    for ( i = 0; i <= 1; i++ ) {
36
37       /* muestra los valores de las columnas */
38       for ( j = 0; j <= 2; j++ ) {
39          printf( “%d “, a[ i ][ j ] );
40       } /* fin del for interno */
41
42       printf( “\n” ); /* comienza la nueva línea de salida */
43    } /* fin del for externo */
44
45 } /* fin de la función imprimeArreglo */
```

```
Los valores en el arreglo1 por linea son:
1 2 3
4 5 6
Los valores en el arreglo2 por linea son:
1 2 3
4 5 0
Los valores en el arreglo3 por linea son:
1 2 0
4 0 0
```

**Figura 6.21** Inicialización de arreglos multidimensionales. (Parte 2 de 2.)

El programa declara tres arreglos de dos filas y tres columnas (cada uno con seis elementos). La declaración de **`arreglo1`** (línea 11) proporciona seis inicializadores en dos sublistas. La primera sublista inicializa la primera fila del arreglo (es decir, la fila 0) con los valores 1, 2 y 3; y la segunda sublista inicializa la segunda fila del arreglo (es decir, la fila 1) con los valores 4, 5 y 6.

Si las llaves alrededor de cada sublista son removidas de **`arreglo1`**, el compilador inicializa los elementos de la primera fila seguido por los elementos de la segunda fila. La definición de **`arreglo2`** (línea 12) proporciona cinco inicializadores. Los inicializadores se asignan a la primera fila y luego a la segunda. Cualquier elemento que no tenga explícitamente un inicializador, se inicializa automáticamente en cero, por lo que **`arreglo2[ 1 ][ 2 ]`** se inicializa en 0.

La declaración de **arreglo3** (línea 13) proporciona tres inicializadores en dos sublistas. La sublista para la primera fila inicializa explícitamente en 1 y 2 a los dos primeros elementos de la primera fila. El tercer elemento se inicializa en 0. La sublista para la segunda fila inicializa explícitamente en 4 al primer elemento. Los dos últimos elementos se inicializan en cero.

El programa llama a la función **despliegaArreglo** (líneas 29-45) para mostrar cada uno de los elementos del arreglo. Observe que la definición de la función especifica el parámetro del arreglo como **const int a[ ][ 3 ]**. Cuando recibimos un arreglo con un solo subíndice como el argumento de una función, los corchetes del arreglo están vacíos en la lista de parámetros de la función. El primer subíndice de un arreglo con múltiples subíndices tampoco es necesario, pero todos los subíndices subsiguientes sí lo son. El compilador utiliza estos subíndices para determinar las posiciones en memoria de los elementos correspondientes a arreglos con múltiples subíndices. Todos los elementos de un arreglo se almacenan en memoria de manera consecutiva, independientemente del número de subíndices. En un arreglo con dos subíndices, la primera fila se almacena en memoria, seguida por la segunda.

Proporcionar los valores de los subíndices en la declaración de un parámetro, permite al compilador indicarle a la función cómo localizar un elemento del arreglo. En un arreglo con dos subíndices, cada fila es básicamente un arreglo con un subíndice. Para localizar un elemento de una fila en particular, el compilador debe saber exactamente cuántos elementos hay en cada fila, para que cuando acceda al arreglo pueda saltar el número adecuado de posiciones de memoria. Entonces, cuando en nuestro ejemplo se accede a **a[ 1 ][ 2 ]**, el compilador sabe que debe saltar los tres elementos de la primera fila en memoria, para llegar a la segunda fila (fila 1). Después, el compilador accede al tercer elemento de esa fila (elemento 2).

Muchas formas comunes de manipulación de arreglos utilizan instrucciones de repetición **for**. Por ejemplo, la siguiente instrucción **for** establece en cero todos los elementos de la tercera fila del arreglo a correspondiente a la figura 6.20:

```
for ( columna = 0; columna < 3; columna++ )
   a[ 2 ][ columna ] = 0;
```

Nosotros especificamos la *tercera* fila y, por lo tanto, sabemos que el primer subíndice siempre es **2** (de nuevo, **0** es la primera fila y **1** es la segunda fila). La instrucción **for** varía sólo el segundo subíndice (es decir, el subíndice de columna). La instrucción **for** anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción **for** anidada determina el total de los elementos del arreglo **a**:

```
total = 0;

for ( fila = 0; fila < 2; fila++ )
   for ( columna = 0; columna <= 3; columna++ )

      total += a[ fila ][ columna ];
```

La instrucción **for** obtiene el total de los elementos del arreglo, una fila a la vez. La instrucción **for** externa comienza por establecer **fila** (es decir, el subíndice de la fila) en **0**, de manera que los elementos de la primera fila pueden sumarse mediante la estructura **for** interna. La instrucción **for** externa incrementa **fila** a **1**, de manera que los elementos de la segunda fila se pueden sumar. Después, la instrucción **for** externa incrementa **fila** a **2**, por lo que los elementos de la tercera fila se pueden sumar. El resultado se imprime cuando las instrucciones **for** anidadas terminan.

La figura 6.22 realiza muchas otras manipulaciones comunes sobre el arreglo 3 por 4, **calificacionesEstudiante** utilizando el comando **for**. Cada fila del arreglo representa a un estudiante, y cada columna representa la calificación de uno de los cuatro exámenes que presentaron durante el semestre. Las manipulaciones al arreglo se realizan por medio de cuatro funciones. La función **minimo** (líneas 44 a 66) determina la calificación más baja de cualquier estudiante durante el semestre. La función **maximo** (líneas 69 a 91) determina la calificación más alta de cualquier estudiante durante el semestre. La función **promedio** (líneas 94 a 106)

determina el promedio particular de cada estudiante durante el semestre. La función **despliegaArreglo** (líneas 109 a 130) despliega claramente el arreglo con dos subíndices en un formato tabular.

```
1  /* Figura 6.22: fig06_22.c
2     Ejemplo de un arreglo de doble subíndice */
3  #include <stdio.h>
4  #define ESTUDIANTES 3
5  #define EXAMENES 4
6
7  /* prototipos de las funciones */
8  int minimo( const int calificaciones[][ EXAMENES ], int alumnos, int
   examenes );
9  int maximo( const int calificaciones[][ EXAMENES ], int alumnos, int
   examenes );
10 double promedio( const int estableceCalif[], int examenes );
11 void despliegaArreglo( const int calificaciones[][ EXAMENES ], int alumnos,
   int examenes );
12
13 /* la función main comienza la ejecución del programa */
14 int main()
15 {
16    int estudiante; /* contador de estudiantes */
17
18    /* inicializa las calificaciones para tres estudiantes (filas) */
19    const int calificacionesEstudiantes[ ESTUDIANTES ][ EXAMENES ] =
20       { { 77, 68, 86, 73 },
21         { 96, 87, 89, 78 },
22         { 70, 90, 86, 81 } };
23
24    /* muestra el arreglo calificacionesEstudiantes */
25    printf( “El arreglo es:\n” );
26    despliegaArreglo( calificacionesEstudiantes, ESTUDIANTES, EXAMENES );
27
28    /* determina el valor más pequeño y el valor más grande de las
         calificaciones */
29    printf( “\n\nCalificacion mas baja: %d\nCalificacion mas alta: %d\n”,
30       minimo( calificacionesEstudiantes, ESTUDIANTES, EXAMENES ),
31       maximo( calificacionesEstudiantes, ESTUDIANTES, EXAMENES ) );
32
33    /* calcula el promedio de calificaciones de cada estudiante */
34    for ( estudiante = 0; estudiante < ESTUDIANTES; estudiante++ ) {
35       printf( “El promedio de calificacion del estudiante %d es %.2f\n”,
36          estudiante, promedio( calificacionesEstudiantes[ estudiante ],
          EXAMENES ) );
37    } /* fin de for */
38
39    return 0; /* indica terminación exitosa */
40
41 } /* fin de main */
42
43 /* Encuentra la calificación mínima */
44 int minimo( const int calificaciones[][ EXAMENES ], int alumnos, int
   examenes )
45 {
```

**Figura 6.22** Ejemplo de arreglos con dos subíndices. (Parte 1 de 3.)

```
   int i; /* contador de estudiantes */
   int j; /* contador de examenes */
   int califBaja = 100; /* inicializa a la calificación más alta posible */

   /* ciclo a través de las filas de calificaciones */
   for ( i = 0; i < alumnos; i++ ) {

      /* ciclo a través de las columnas de calificaciones */
      for ( j = 0; j < examenes; j++ ) {

         if ( calificaciones[ i ][ j ] < califBaja ) {
            califBaja = calificaciones[ i ][ j ];
         } /* fin de if */

      } /* fin del for interno */

   } /* fin del for externo */

   return califBaja; /* devuelve la calificación mínima */

} /* fin de la función main */

/* Encuentra la calificación más alta */
int maximo( const int calificaciones[][ EXAMENES ], int alumnos, int examenes )
{
   int i; /* contador de estudiantes */
   int j; /* contador de examenes */
   int califAlta = 0; /* inicializa a la calificación más baja posible */

   /* ciclo a través de las filas de calificaciones */
   for ( i = 0; i < alumnos; i++ ) {

      /* ciclo a través de las columnas de calificaciones */
      for ( j = 0; j < examenes; j++ ) {

         if ( calificaciones[ i ][ j ] > califAlta ) {
            califAlta = calificaciones[ i ][ j ];
         } /* fin de if */

      } /* fin del for interno */

   } /* fin del for externo */

   return califAlta; /* devuelve la calificación máxima */

} /* fin de la función maximo */

/* Determina la calificación promedio para un estudiante en especial */
double promedio( const int conjuntoDeCalificaciones[], int examenes )
{
   int i; /* contador de exámenes */
   int total = 0; /* suma de las calificaciones del examen */

```

**Figura 6.22** Ejemplo de arreglos con dos subíndices. (Parte 2 de 3.)

```
   /* total de calificaciones de un estudiante */
   for ( i = 0; i < examenes; i++ ) {
      total += conjuntoDeCalificaciones[ i ];
   } /* fin de for */

   return ( double ) total / examenes; /* promedio */

} /* fin de la función promedio */

/* Imprime el arreglo */
void despliegaArreglo( const int calificaciones[][ EXAMENES ], int alumnos,
int examenes )
{
   int i; /* contador de estudiantes */
   int j; /* contador de examenes */

   /* muestra el encabezado de las columnas */
   printf( “                  [0]  [1]  [2]  [3]” );

   /* muestra las calificaciones en forma tabular */
   for ( i = 0; i < alumnos; i++ ) {

      /* muestra la etiqueta de la fila */
      printf( “\ncalificacionesEstudiantes[%d] “, i );

      /* muestra las calificaciones de un estudiante */
      for ( j = 0; j < examenes; j++ ) {
         printf( “%-5d”, calificaciones[ i ][ j ] );
      } /* fin del for interno */

   } /* fin del for externo */

} /* fin de la función despliegaArreglo */
```

```
El arreglo es:
                             [0]  [1]  [2]  [3]
calificacionesEstudiantes[0] 77   68   86   73
calificacionesEstudiantes[1] 96   87   89   78
calificacionesEstudiantes[2] 70   90   86   81

Calificacion mas baja: 68
Calificacion mas alta: 96
El promedio de calificacion del estudiante 0 es 76.00
El promedio de calificacion del estudiante 1 es 87.50
El promedio de calificacion del estudiante 2 es 81.75
```

**Figura 6.22** Ejemplo de arreglos con dos subíndices. (Parte 3 de 3.)

Las funciones **`minimo`**, **`maximo`** y **`despliegaArreglo`** reciben, cada una, tres argumentos: el arreglo **`calificacionesEstudiante`** (llamado **`calificaciones`** en cada función), el número de estudiantes (las filas en el arreglo), y el número de exámenes (las columnas del arreglo). Cada función realiza un ciclo a

través del arreglo **calificaciones**, utilizando instrucciones **for** anidadas. La siguiente instrucción **for** anidada corresponde a la definición de la función **minimo**:

```
/* ciclo a través de las filas de calificaciones */
for ( i = 0; i < alumnos; i++ ) {
   /* ciclo a través de las columnas de calificaciones */
      for ( j = 0; j < examenes; j++ ) {

      if ( calificaciones[ i ][ j ] < califBaja ) {
         califBaja = calificaciones[ i ][ j ];

      } /* fin de if */
   } /* fin del for interno */
} /* fin del for externo */
```

La instrucción **for** externa comienza al establecer **i** (es decir, el subíndice de fila) en **0**, así, los elementos de la primera fila se pueden comparar con la variable **califBaja** del cuerpo de la instrucción **for** interna. La instrucción **for** interna realiza un ciclo a través de las cuatro calificaciones de una fila en especial, y compara cada calificación con **califBaja**. Si una calificación es menor que **califBaja**, ésta se establece en dicha calificación. La instrucción **for** externa incrementa el subíndice de la fila a **1**. Los elementos de la segunda fila se comparan con la variable **califBaja**. La instrucción **for** externa incrementa el subíndice a **2**. Los elementos de la tercera fila se comparan con la variable **califBaja**. Cuando la ejecución de la estructura anidada está completa, **califBaja** contiene la calificación más baja del arreglo con dos subíndices. La función **maximo** trabaja de manera similar a la función **minimo**.

La función **promedio** (línea 63) toma dos argumentos: un arreglo con un solo subíndice llamado **conjuntoDeCalificaciones**, el cual contiene los resultados de un estudiante en particular, y el número de resultados de examen en el arreglo. Cuando se llama a **promedio**, se pasa el primer argumento **calificacionesEstudiante[ estudiante ]**. Esto ocasiona que la dirección de una fila del arreglo con dos subíndices pase a **promedio**. El argumento **calificacionesEstudiante[1]** es la dirección con la que comienza la segunda fila del arreglo. Recuerde que un arreglo con dos subíndices es básicamente un arreglo formado por arreglos con un solo subíndice, y que el nombre de un arreglo con un solo subíndice es la dirección en memoria de ese arreglo. La función **promedio** calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de examen y devuelve el resultado de punto flotante.

## RESUMEN

- C almacena listas de valores en arreglos. Un arreglo es un grupo de posiciones de memoria relacionadas. Estas posiciones están relacionadas por el hecho de que tienen el mismo nombre y el mismo tipo. Para hacer referencia a una posición o elemento en particular del arreglo, especificamos el nombre del arreglo y el subíndice.
- Un subíndice puede ser un entero o una expresión entera. Si un programa utiliza una expresión como subíndice, entonces la expresión se evalúa para determinar el elemento particular del arreglo.
- Es importante notar la diferencia cuando se hace referencia al séptimo elemento del arreglo y cuando se hace referencia al elemento siete. El séptimo elemento tiene un subíndice de **6**, mientras que el elemento siete tiene un subíndice de **7** (en realidad el octavo elemento del arreglo). Ésta es una fuente de errores de desplazamiento en uno.
- Los arreglos ocupan espacio en memoria. Para reservar 100 elementos para el arreglo entero **b** y 27 elementos para el arreglo entero **x**, el programador escribe

```
int b[ 100 ], x[ 27 ];
```

- Un arreglo de tipo **char** puede utilizarse para almacenar una cadena de caracteres.
- Los elementos de un arreglo pueden inicializarse en una declaración, en instrucciones de asignación, e introduciendo directamente los valores en los elementos del arreglo.
- Si hay menos inicializadores que elementos en el arreglo, C inicializa los elementos sobrantes en cero.
- C no evita que se haga referencia a elementos que se encuentran fuera de los límites de un arreglo.
- Un arreglo de caracteres puede inicializarse mediante una literal de cadena.
- Todas las cadenas en C finalizan con el carácter nulo. El carácter constante que representa el carácter nulo es **'\0'**.
- Los arreglos de caracteres pueden inicializarse con caracteres constantes en una lista de inicialización.

- Se puede acceder de manera directa a los caracteres individuales de una cadena almacenada en un arreglo, mediante la notación de subíndices para arreglos.
- Desde el teclado es posible introducir una cadena en un arreglo de caracteres, utilizando **`scanf`** y el especificador de conversión **`%s`**.
- Es posible desplegar un arreglo de caracteres que representa una cadena, mediante **`printf`** y el especificador de conversión **`%s`**.
- Aplique **`static`** a la definición de un arreglo local para que el arreglo no se cree cada vez que se llame a la función, y para que el arreglo no se destruya cada vez que la función salga.
- Los arreglos que son **`static`** se inicializan automáticamente en tiempo de compilación. Si el programador no inicializa explícitamente un arreglo **`static`**, el compilador lo inicializa en cero.
- Para pasar un arreglo a una función, se pasa el nombre del arreglo. Para pasar un elemento particular de un arreglo a una función, simplemente pase el nombre del arreglo seguido por el subíndice [que se encuentra entre corchetes] del elemento particular.
- C pasa por referencia los arreglos a funciones; las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de la función que llama. El nombre del arreglo en realidad es la dirección del primer elemento del arreglo. Debido a que la dirección inicial del arreglo es pasada, la función llamada sabe precisamente el lugar en donde está almacenado el arreglo.
- Para recibir un arreglo como argumento, la lista de parámetros de la función debe especificar que se recibirá un arreglo. El tamaño del arreglo no es necesario en los corchetes del arreglo (en el caso de los arreglos con un solo subíndice).
- Cuando se utiliza con **`printf`**, el especificador de conversión **`%p`** normalmente despliega las direcciones como números hexadecimales, pero esto depende de la plataforma.
- C proporciona el calificador especial de tipo **`const`** para evitar modificaciones a los valores de un arreglo en una función. Cuando un parámetro de arreglo es precedido por el calificador **`const`**, los elementos del arreglo se vuelven constantes en el cuerpo de la función, e intentar modificarlos dará como resultado un error en tiempo de compilación.
- Podemos ordenar un arreglo, utilizando la técnica de ordenamiento burbuja. Se hacen diversas pasadas al arreglo. En cada pasada, se comparan pares sucesivos de elementos. Si un par se encuentra en orden (o si los valores son idénticos), se deja así. Si un par está en desorden, los valores se intercambian. Para arreglos pequeños, el ordenamiento burbuja es aceptable, pero para arreglos grandes funciona de manera ineficiente comparado con otros algoritmos más sofisticados de ordenamiento.
- La búsqueda lineal compara cada elemento del arreglo con la clave de búsqueda. Debido a que el arreglo no se encuentra en un orden particular, la probabilidad de que el valor se encuentre en el primer elemento o en el último es la misma. Por lo tanto, en promedio, el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del arreglo. La búsqueda lineal funciona bien para arreglos pequeños o para arreglos desordenados.
- El algoritmo de la búsqueda binaria elimina la mitad de los elementos de un arreglo ordenado después de cada comparación. El algoritmo localiza el elemento central del arreglo y lo compara con la clave de búsqueda. Si son iguales, se localizó la clave de búsqueda y se devuelve el subíndice de ese elemento. Si no son iguales, el problema se reduce a buscar en una mitad del arreglo.
- En el peor de los casos, por medio de la búsqueda binaria, la búsqueda en un arreglo de 1023 elementos tomará sólo 10 comparaciones. En un arreglo de 1048576 ($2^{20}$) elementos tomará un máximo de 20 comparaciones para encontrar la clave. En un arreglo de 1000 millones de elementos tomará un máximo de 30 comparaciones para encontrar la clave.
- Los arreglos pueden utilizarse para representar tablas de valores que consisten en información acomodada en filas y columnas. Para identificar un elemento en particular de una tabla, se especifican dos subíndices: el primero (por convención) identifica la fila en la que el elemento se encuentra, y el segundo (por convención) identifica la columna en la que se encuentra el elemento. Las tablas o arreglos que requieren dos subíndices para identificar un elemento en particular son conocidos como arreglos con dos subíndices.
- Un arreglo con múltiples subíndices puede inicializarse cuando se declara mediante una lista de inicialización.
- Cuando una función recibe como argumento a un arreglo con un solo subíndice, los corchetes del arreglo se encuentran vacíos en la lista de parámetros de la función. El primer subíndice de un arreglo con múltiples subíndices tampoco es necesario, pero los subíndices subsiguientes sí lo son. El compilador utiliza estos subíndices para determinar las posiciones en memoria de los elementos correspondientes a arreglos con múltiples subíndices.
- Para pasar una fila de un arreglo con dos subíndices a una función que recibe como argumento a un arreglo con un solo subíndice, simplemente pase el nombre del arreglo, seguido por el subíndice [entre corchetes] de esa fila.

## TERMINOLOGÍA

**`a[ i ]`**
**`a[ i ][ j ]`**
análisis de los datos de una encuesta
anulación de un arreglo
área temporal para el intercambio de valores
arreglo
arreglo con dos subíndices
arreglo con múltiples subíndices
arreglo con un solo subíndice
arreglo de caracteres
arreglo de *m* por *n*
búsqueda binaria
búsqueda en un arreglo
búsqueda lineal
cadena
calificador **`const`**
carácter de terminación nulo
carácter nulo **`'\0'`**
clave de búsqueda
constante simbólica
corchetes
declaración de un arreglo
directiva de preprocesador **`#define`**
elemento cero
elemento de un arreglo
error de desplazamiento en uno
escalable
escalar
especificador de conversión **`%p`**
expresión como subíndice
formato tabular
gráfico de barras
inicialización de un arreglo
lista de inicializadores
lista de inicializadores de arreglos
media
mediana
moda
nombre de un arreglo
ordenamiento
ordenamiento burbuja
ordenamiento de los elementos de un arreglo
ordenamiento por hundimiento
pasada de ordenamiento
paso de arreglos a funciones
paso por referencia
posición numérica
precisión doble
subíndice
subíndice de columna
subíndice de fila
suma de los elementos de un arreglo
tabla de valores
texto de reemplazo
valor de un elemento
verificación de límites

## ERRORES COMUNES DE PROGRAMACIÓN

**6.1** Es importante notar la diferencia entre el "séptimo elemento del arreglo" y el "elemento siete del arreglo". Debido a que los subíndices de los arreglos comienzan en 0, el "séptimo elemento del arreglo" tiene un subíndice de 6, mientras que "el elemento siete del arreglo" tiene un subíndice de 7 y, en realidad, es el octavo elemento del arreglo. Ésta es una fuente de "errores de desplazamiento en uno".

**6.2** Olvidar inicializar los elementos de un arreglo, cuyos elementos debieran inicializarse.

**6.3** Proporcionar más inicializadores en una lista de inicialización que elementos en el arreglo, es un error de sintaxis.

**6.4** Finalizar una directiva de preprocesador **`#define`** o **`#include`** con un punto y coma. Recuerde que las directivas de preprocesador no son instrucciones de C.

**6.5** Asignar un valor a una constante simbólica en una instrucción ejecutable, es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio alguno para ella, como lo hace con las variables que contienen valores en tiempo de ejecución.

**6.6** Hacer referencia a un elemento que se encuentra fuera de los límites del arreglo.

**6.7** No proporcionarle a **`scanf`** un arreglo de caracteres lo suficientemente grande para almacenar una cadena escrita mediante el teclado, puede ocasionar la destrucción de los datos de un programa y otros errores en tiempo de ejecución.

**6.8** Suponer que los elementos de un arreglo local **`static`** se inicializan en cero cada vez que se llama a la función en la que el arreglo está declarado.

**6.9** Hacer referencia a un elemento de un arreglo con dos subíndices de la forma **`a[ x, y ]`**, en lugar de hacerlo de la forma **`a[ x ][ y ]`**.

## TIPS PARA PREVENIR ERRORES

**6.1** Cuando se hace un ciclo en torno a un arreglo, el subíndice del arreglo nunca debe ser menor que 0 y siempre debe ser menor que el número total de elementos del arreglo (tamaño –1). Asegúrese que la condición de terminación de ciclo prevenga el acceso de elementos fuera de este rango.

**6.2** Los programas deben validar que todos los valores de entrada sean correctos, para evitar que información errónea afecte los cálculos del programa.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**6.1** Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y recuerda al programador que las constantes simbólicas no son variables.

**6.2** En nombres de constantes simbólicas que contengan varias palabras, utilice guiones bajos para separarlas y, así, mejorar su legibilidad.

**6.3** Busque la claridad de los programas. A veces, vale la pena perder un poco de eficiencia en cuanto al uso de la memoria o del procesador, a favor de la creación de programas más claros.

## TIPS DE RENDIMIENTO

**6.1** En ocasiones, las consideraciones relacionadas con el rendimiento se alejan demasiado de las consideraciones para lograr la claridad.

**6.2** En funciones que contienen arreglos automáticos, en donde la función entra y sale con frecuencia del alcance, haga que el arreglo sea **`static`** para que éste no se genere cada vez que se invoque a la función.

**6.3** Pasar arreglos por referencia tiene sentido por motivos de rendimiento. Si los arreglos se pasaran por valor, entonces una copia de cada elemento también pasaría. Esto implicaría que para pasar arreglos grandes y de manera frecuente, se requeriría demasiado tiempo y demasiado espacio de almacenamiento para las copias de los arreglos.

**6.4** Algunas veces, los algoritmos más sencillos tienen un rendimiento muy pobre. Su virtud radica en que son fáciles de escribir, probar y depurar. Sin embargo, los algoritmos más complejos son necesarios para lograr un máximo rendimiento.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**6.1** Definir el tamaño de un arreglo como una constante simbólica hace que los programas sean más escalables.

**6.2** Es posible pasar un arreglo por valor (mediante un simple truco que explicaremos en el capítulo 10).

**6.3** El calificador de tipo **`const`** puede aplicarse a un parámetro de arreglo en la declaración de una función, para prevenir que el arreglo original sea modificado en el cuerpo de la función. Éste es otro ejemplo del principio del menor privilegio. A las funciones no se les debe dar la capacidad de modificar un arreglo, a menos que sea absolutamente necesario.

## EJERCICIOS DE AUTOEVALUACIÓN

**6.1** Complete los espacios en blanco:

a) Las listas y las tablas de valores se almacenan en ______.

b) Los elementos de un arreglo se relacionan por el hecho de que tienen el mismo ______ y ______.

c) Al número utilizado para hacer referencia a un elemento particular de un arreglo se le llama ______.

d) Debe utilizarse una ______ para especificar el tamaño de un arreglo, debido a que ésta hace al programa más escalable.

e) Al proceso de colocar en orden a los elementos de un arreglo se le llama ______ de un arreglo.

f) Determinar si un arreglo contiene un cierto valor clave se le llama ______ en el arreglo.

g) A un arreglo que utiliza dos subíndices se le conoce como arreglo ______.

**6.2** Diga cuáles de los siguientes enunciados son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.

a) Un arreglo puede almacenar muchos tipos diferentes de valores.

b) El subíndice de un arreglo puede ser del tipo de datos **`double`**.

c) Si en una lista de inicializadores existen menos inicializadores que elementos del arreglo, C inicializa automáticamente con el último valor de la lista a los elementos sobrantes.

d) Si una lista de inicializadores contiene más inicializadores que elementos en el arreglo, es un error.

e) Un elemento particular de un arreglo que es pasado a una función y modificado en la función llamada, contendrá el valor modificado en la función que llama.

**6.3** Responda las siguientes preguntas, con respecto a un arreglo llamado **`fracciones`**.

a) Declare una constante simbólica **`TAMANIO`** para que sea reemplazada con el texto de reemplazo 10.

b) Declare un arreglo con **`TAMANIO`** elementos de tipo **`double`**, e inicialice los elementos en 0.

c) Asigne un nombre al cuarto elemento del arreglo.

d) Haga referencia al elemento 4 del arreglo.
e) Asigne el valor **1.667** al elemento 9 del arreglo.
f) Asigne el valor **3.333** al séptimo elemento del arreglo.
g) Despliegue los elementos 6 y 9 del arreglo con dos dígitos de precisión a la derecha del punto decimal, y muestre la salida que aparece en pantalla.
h) Despliegue todos los elementos del arreglo mediante la instrucción de repetición **for**. Suponga que una variable entera **x** ha sido definida como una variable de control para el ciclo. Muestre la salida.

**6.4** Escriba instrucciones que realicen lo siguiente:
a) Declare **tabla** para que sea un arreglo entero y que tenga 3 filas y 3 columnas. Suponga que la constante simbólica **TAMANIO** se declaró para que fuera 3.
b) ¿Cuántos elementos contiene el arreglo **tabla**? Imprima el número total de elementos.
c) Utilice una instrucción de repetición **for** para inicializar cada elemento de **tabla** con la suma de sus subíndices. Suponga que las variables enteras **x** y **y** se definieron como variables de control.
d) Imprima los valores de cada elemento del arreglo **tabla**. Suponga que el arreglo se inicializó con la declaración:

```
int tabla[ TAMANIO ][ TAMANIO ] =
    { { 1, 8}, { 2, 4, 6 }, { 5 } };
```

**6.5** Encuentre el error en cada uno de los siguientes segmentos de programa y corríjalo:
a) **#define TAMANIO 100;**
b) **TAMANIO = 10;**
c) *Suponga que* **int b[ 10 ] = { 0 }, i;**

```
for ( i = 0; i <= 10; i++ )
    b[ i ] = 1;
```

d) **#include <stdio.h>;**
e) *Suponga que* **int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };**

```
    a[ 1, 1 ] = 5;
```

f) **#define VALOR = 120**

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**6.1** a) Arreglos. b) Nombre, tipo. c) Subíndice. d) Constante simbólica. e) Ordenamiento. f) Búsqueda. g) Con dos subíndices.

**6.2** a) Falso. Un arreglo puede almacenar sólo valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Falso. C inicializa automáticamente en cero a los elementos sobrantes.
d) Verdadero.
e) Falso. Los elementos individuales de un arreglo se pasan por valor. Si el arreglo completo se pasa a una función, entonces cualquier modificación se reflejará en la original.

**6.3** a) **#define TAMANIO 10**
b) **double fracciones[ TAMANIO ] = { 0 };**
c) **fracciones[ 3 ]**
d) **fracciones[ 4 ]**
e) **fracciones[ 9 ] = 1.667;**
f) **fracciones[ 6 ] = 3.333;**
g) **printf( “%.2f %.2f\n”, fracciones[ 6 ], fracciones[ 9 ] );**
*Salida*: **3.33 1.67.**
h) **for ( x = 0; x < TAMANIO; x++ )**

```
  printf( "fracciones[%d] = %f\n", fracciones[ x ] );
```

*Salida*:

```
fracciones [ 0 ] = 0.000000
fracciones [ 1 ] = 0.000000
fracciones [ 2 ] = 0.000000
fracciones [ 3 ] = 0.000000
```

```
fracciones [ 4 ] = 0.000000
fracciones [ 5 ] = 0.000000
fracciones [ 6 ] = 3.333000
fracciones [ 7 ] = 0.000000
fracciones [ 8 ] = 0.000000
fracciones [ 9 ] = 1.667000
```

**6.4** a) `int tabla[ TAMANIO ][ TAMANIO ];`

b) Nueve elementos. `printf( "%d\n", TAMANIO * TAMANIO );`

c)
```
for ( x = 0; x < TAMANIO; x++ )

   for ( y = 0; t < TAMANIO; y++ )
      tabla[ x ][ y ] = x + y;
```

d)
```
for ( x = 0; x < TAMANIO; x++ )
      for ( y = 0; y < TAMANIO; y++ )
            printf( "tabla[%d][%d] = %d\n", x, y, tabla[ x ][ y ] );
```

*Salida*:
```
tabla[0][0] = 1
tabla[0][1] = 8
tabla[0][2] = 0
tabla[1][0] = 2
tabla[1][1] = 4
tabla[1][2] = 6
tabla[2][0] = 5
tabla[2][1] = 0
tabla[2][2] = 0
```

**6.5** a) Error: punto y coma al final de la directiva de preprocesador `#define`.
Corrección: eliminar el punto y coma.

b) Error: asignar un valor a una constante simbólica mediante una instrucción de asignación.
Corrección: asignar un valor a la constante simbólica en una directiva de preprocesador `#define`, sin utilizar el operador de asignación como en `#define TAMANIO 10`.

c) Error: hacer referencia a un elemento del arreglo fuera de los límites del arreglo (`b[10]`).
Corrección: modifique el valor final de la variable de control a `9`.

d) Error: punto y coma al final de la directiva de preprocesador `#include`.
Corrección: elimine el punto y coma.

e) Error: subíndices incorrectos en el arreglo.
Corrección: modifique la instrucción a `a[ 1 ][ 1 ] = 5`;

f) Error: asignar un valor a una constante simbólica mediante una instrucción de asignación.
Corrección: asigne un valor a la constante simbólica en una directiva de preprocesador `#define`, sin utilizar el operador de asignación como en `#define VALOR 120`.

## *EJERCICIOS*

**6.6** Complete los espacios en blanco:

a) C almacena listas de valores en ____________.

b) Los elementos de un arreglo están relacionados por el hecho de que ____________.

c) Cuando se hace referencia a un elemento de un arreglo, la posición numérica contenida entre corchetes se llama ____________.

d) Los nombres de los cinco elementos del arreglo **p** son ____________, ____________, ____________, ____________ y ____________.

e) El contenido de un elemento particular de un arreglo se conoce como el ____________ de ese elemento.

f) Asignar un nombre a un arreglo, establecer su tipo y especificar el número de elementos en el arreglo se conoce como ____________ al arreglo.

g) Al proceso de colocar los elementos de un arreglo en un orden ascendente o descendente se le conoce como ____________.

h) En un arreglo con dos subíndices, el primer subíndice (por convención) identifica la ____________ de un elemento, y el segundo subíndice (por convención) identifica la ____________ de un elemento.

i) Un arreglo de *m* por *n* contiene ____________ filas, ____________ columnas y ____________ elementos.

j) El nombre del elemento que se encuentra en la fila 3 y columna 5 del arreglo **d** es ____________.

**6.7** Diga cuáles de los siguientes enunciados son *verdaderos* y cuáles son *falsos*. Si la respuesta es *falso*, explique por qué.

a) Para hacer referencia a una ubicación en particular de memoria dentro de un arreglo, especificamos el nombre del arreglo y el valor de un elemento en particular.
b) Una declaración de arreglo reserva espacio para el arreglo.
c) Para indicar que se deben reservar 100 ubicaciones para un arreglo entero **p**, el programador escribe la declaración
**p[ 100 ];**
d) Un programa en C que inicializa en cero a los elementos de un arreglo de 15 elementos debe contener una instrucción **for**.
e) Un programa en C que suma el número de elementos de un arreglo con dos subíndices, debe contener instrucciones **for** anidadas.
f) La media, mediana y moda del siguiente conjunto de valores son 5, 6 y 7, respectivamente: 1, 2, 5, 6, 7, 7, 7.

**6.8** Escriba las instrucciones para llevar a cabo cada una de las siguientes tareas:

a) Despliegue el valor del séptimo elemento del arreglo de caracteres **f**.
b) Introduzca un valor en el elemento 4 del arreglo de punto flotante con un solo subíndice, **b**.
c) Inicialice en 8 cada uno de los 5 elementos del arreglo entero **g**.
d) Sume los elementos del arreglo de punto flotante **c**, el cual contiene 100 elementos.
e) Copie el arreglo **a** en la primera porción del arreglo **b**. Suponga que **double a[ 11 ]**, **b[ 34 ]**;
f) Determine y despliegue los valores más pequeño y más grande contenidos en el arreglo de punto flotante **w**, de 99 elementos.

**6.9** Considere el arreglo entero **t** de 2 por 5.

a) Escriba la declaración para **t**.
b) ¿Cuántas filas tiene **t**?
c) ¿Cuántas columnas tiene **t**?
d) ¿Cuántos elementos tiene **t**?
e) Escriba los nombres de todos los elementos que se encuentran en la segunda fila de **t**.
f) Escriba los nombres de todos los elementos que se encuentran en la tercera columna de **t**.
g) Escriba una instrucción que establezca en cero el elemento de la fila 1 y la columna 2 de **t**.
h) Escriba una serie de instrucciones que inicialice en cero cada elemento de **t**. No utilice una estructura de repetición.
i) Escriba una instrucción **for** anidada que inicialice en cero cada elemento de **t**.
j) Escriba una instrucción que introduzca los valores para los elementos de **t** desde la terminal.
k) Escriba una serie de instrucciones que determine y despliegue el valor más pequeño del arreglo **t**.
l) Escriba una instrucción que despliegue los elementos de la primera fila de **t**.
m) Escriba una instrucción que sume los elementos de la cuarta columna de **t**.
n) Escriba una serie de instrucciones que despliegue el arreglo **t** en un formato tabular. Liste los subíndices de columna como encabezados horizontales y los subíndices de fila a la derecha de cada fila.

**6.10** Utilice un arreglo con un solo subíndice para resolver el siguiente problema. Una empresa paga a su personal de ventas con base en una comisión. El personal de ventas recibe $200 por semana, más 9 por ciento de sus ventas totales semanales. Por ejemplo, un vendedor que suma $3000 en ventas semanales recibe $200 más el 9 por ciento de $3000, o un total de $470. Escriba un programa en C (que utilice un arreglo de contadores) que determine cuántos de los vendedores reciben salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca para obtener un monto entero):

a) de $200 a $299
b) de $300 a $399
c) de $400 a $499
d) de $500 a $599
e) de $600 a $699
f) de $700 a $799
g) de $800 a $899
h) de $900 a $999
i) de $1000 o más

**6.11** El ordenamiento de burbuja que presentamos en la figura 6.15 es ineficiente para arreglos grandes. Haga las siguientes modificaciones sencillas, para mejorar el rendimiento del ordenamiento de burbuja:

a) Después de la primera pasada, seguramente el número más alto es el elemento más grande del arreglo; después de la segunda pasada, los dos números más altos se encuentran "en su lugar", y así sucesivamente. En lugar de

hacer nueve comparaciones en cada pasada, modifique el programa de ordenamiento de burbuja para hacer ocho comparaciones en la segunda pasada, siete en la tercera pasada, y así sucesivamente.

b) Los datos en el arreglo pudieran estar ya en el orden apropiado o cerca del orden apropiado, ¿entonces, por qué hacer nueve pasadas si con menos podría ser suficiente? Modifique el ordenamiento para verificar, al final de cada pasada, si se han hecho intercambios. Si no se han hecho intercambios, entonces los datos deben estar ya en el orden apropiado, de manera que el programa debe terminar. Si se hicieron intercambios, entonces se requiere al menos una pasada.

**6.12** Escriba instrucciones individuales que realicen cada una de las siguientes operaciones correspondientes a arreglos con un solo subíndice:

a) Inicialice en cero los 10 elementos del arreglo entero **`cuentas`**.
b) Sume 1 a cada uno de los 15 elementos del arreglo entero **`bonos`**.
c) Lea los 12 valores introducidos desde el teclado del arreglo de punto flotante **`temeperaturasCadaMes`**.
d) Despliegue en formato de columnas los 5 valores del arreglo entero **`mejoresMarcas`**.

**6.13** Encuentre el(los) error(es) en cada una de las siguientes instrucciones:

a) Suponga que: **`char str[ 5 ];`**
```
scanf( “%s”, str ); /* El usuario escribe hola */
```
b) Suponga que: **`int a[ 3 ];`**
```
printf( “$d  %d  %d\n”, a[ 1 ], a[ 2 ], a[ 3 ] );
```
c) **`double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`**
d) Suponga que: **`double d[ 2 ][ 10 ];`**
```
d[ 1, 9 ] = 2.345;
```

**6.14** Modifique el programa de la figura 6.16 para que la función moda sea capaz de manipular un empate para el valor de la moda. Además, modifique la función **`mediana`** de manera que los dos elementos centrales sean promediados en un arreglo con un número par de elementos.

**6.15** Utilice un arreglo con un solo subíndice para resolver el siguiente problema. Lea 20 números, en donde cada uno se encuentre entre 10 y 100, inclusive. Mientras se lee cada número, despliéguelo solamente si no es un duplicado de un número ya leído. Prevenga el “peor de los casos”, en el cual los veinte números sean diferentes. Utilice el menor tamaño posible del arreglo para resolver este problema.

**6.16** Etiquete los elementos del arreglo ventas (el cual es un arreglo con dos subíndices de 3 por 5) para indicar el orden en el cual se establecen en cero, con el siguiente segmento de programa:

```
for ( fila = 0; fila < 2; fila++ )
   for ( columna = 0; columna < 4; columna++ )
   ventas[ fila ][ columna ] = 0;
```

**6.17** ¿Qué hace el siguiente programa?

```
/* ej06_17.c */
/* ¿Qué hace este programa? */
#include <stdio.h>
#define TAMANIO 10

int queEsEsto( const int b[], int p ); /* prototipo de la función */

/* la función main comienza la ejecución del programa */
int main()
{
   int x; /* almacena el valor de retorno de la función queEsEsto */

   /* inicializa el arreglo a */
   int a[ TAMANIO ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   x = queEsEsto( a, TAMANIO );

   printf( “El resultado es %d\n”, x );

```

(Parte 1 de 2.)

```
   return 0; /* indica terminación exitosa */

} /* fin de main */

/* ¿Qué hace esta función? */
int queEsEsto( const int b[], int p )
{
   /* caso base */
   if ( p == 1 ) {
      return b[ 0 ];
   } /* fin de if */
   else { /* paso recursivo */

      return b[ p - 1 ] + queEsEsto( b, p - 1 );
   } /* fin de else */

} /* fin de la función queEsEsto */
```

(Parte 2 de 2.)

**6.18** ¿Qué hace el siguiente programa?

```
/* ej06_18.c */
/* ¿Qué hace este programa?  */
#include <stdio.h>
#define TAMANIO 10

/* prototipo de la función */
void algunaFuncion( const int b[], int comienzaIndice, int tamanio );

/* la función main comienza la ejecución del problema */
int main()
{
   int a[ TAMANIO ] = { 8, 3, 1, 2, 6, 0, 9, 7, 4, 5 }; /* inicializa a */

   printf( "La respuesta es:\n" );
   algunaFuncion( a, 0, TAMANIO );
   printf( "\n" );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* ¿Qué hace esta función? */
void algunaFuncion( const int b[], int iniciaIndice, int tamanio )
{
   if ( iniciaIndice < tamanio ) {
      algunaFuncion( b, iniciaIndice + 1, tamanio );
      printf( "%d  ", b[ iniciaIndice ] );
   } /* fin de if */

} /* fin de la función algunaFuncion */
```

**6.19** Escriba un programa que simule el tiro de dos dados. El programa debe utilizar `rand` para tirar el primer dado, y debe utilizar rand de nuevo para tirar el segundo dado. Después, se debe calcular la suma de los dos valores.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Figura 6.23** Salidas del tiro de dados.

[*Nota:* Cada dado puede mostrar un valor entero de 1 a 6, de manera que la suma de los dos valores puede variar de 2 a 12, donde 7 es la suma más frecuente y 2 y 12 son las sumas menos frecuentes]. La figura 6.23 muestra las 36 combinaciones posibles de los dos dados. Su programa debe lanzar los dos dados 36,000 veces. Utilice un arreglo con un solo subíndice para registrar el número de veces que aparece cada suma posible. Despliegue los resultados en formato tabular. Además, determine si los totales son razonables (es decir, existen seis maneras de tirar un 7, de manera que aproximadamente un sexto de todos los tiros debe ser 7).

**6.20** Escriba un programa que ejecute 1000 juegos de craps (sin intervención humana) y responda las siguientes preguntas

a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo tiro, …, en el tiro número 20, y después del tiro número 20?
b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo tiro, …, en el tiro número 20, y después del tiro número 20?
c) ¿Cuál es la oportunidad de ganar en craps? (*Nota:* Usted debe saber que craps es uno de los juegos de casino más imparciales. ¿Qué cree usted que significa esto?
d) ¿Cuál es la duración promedio de un juego de craps?
e) ¿Mejoran las oportunidades de ganar con la duración del juego?

**6.21** (*Sistema de reservaciones para una aerolínea.*) Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema automático de reservaciones. A usted se le ha pedido que programe el nuevo sistema. Usted debe escribir un programa que asigne los asientos, en cada vuelo, del único avión de la aerolínea (capacidad: 10 asientos). Su programa debe desplegar el siguiente menú de alternativas:

```
Por favor, digite 1 para “primera clase”
Por favor, digite 2 para “económico”
```

Si la persona digita 1, su programa debe asignar un asiento en la sección de primera clase (asientos 1 a 5). Si la persona digita 2, su programa debe asignar un asiento en la sección económica (asientos 6 a 10). Su programa debe imprimir un pase de abordar que indique el número de asiento de la persona y si está en la sección de primera clase o en la sección económica del avión.

Utilice un arreglo con un solo subíndice para representar la tabla de asientos del avión. Inicialice en 0 todos los elementos del arreglo para indicar que todos los asientos están vacíos. Mientras se asigna cada asiento, el valor de los elementos correspondientes del arreglo se establece en 1, para indicar que el asiento ya no está disponible.

Por supuesto, su programa nunca debe asignar un asiento que ya está asignado. Cuando la sección de primera clase está llena, su programa debe preguntar a la persona si acepta que se le coloque en la sección económica (y viceversa). Si acepta, entonces haga la asignación apropiada del asiento. Si no acepta, entonces despliegue el mensaje **`“El siguiente vuelo parte en tres horas”`**.

**6.22** Utilice un arreglo con doble subíndice para resolver el siguiente problema. Una empresa tiene cuatro vendedores (1 a 4) los cuales venden cinco productos diferentes (1 a 5). Una vez al día, cada vendedor introduce un registro para cada tipo de producto vendido. Cada registro contiene lo siguiente:

a) El número de vendedor.
b) El número de producto.
c) El monto total del producto vendido del día.

Por lo tanto, cada vendedor pasa entre 0 y 5 registros al día. Suponga que están disponibles los registros con la información del último mes. Escriba un programa que lea toda esta información de las ventas del último mes y sume el total de ventas por vendedor y por producto. Todos los totales se deben almacenar en el arreglo con dos subíndices, ventas. Una vez procesada toda la información del último mes, despliegue los resultados en formato tabular

en donde cada una de las columnas representa a un vendedor y cada una de las filas representa un producto en particular. Obtenga la suma de cada fila para el total de ventas de cada producto del último mes; obtenga la suma de cada columna para el total de ventas por vendedor del último mes. Su salida tabular debe incluir estos totales a la derecha para las filas y en el fondo para las columnas.

**6.23** (*Gráficos de tortuga.*) El lenguaje Logo, que es especialmente popular entre los usuarios de computadoras personales, hizo famoso el concepto de los *gráficos de tortuga*. Imagine una tortuga mecánica que camina alrededor de una habitación bajo el control de un programa en C. La tortuga mantiene una pluma en una de dos posiciones: arriba o abajo. Mientras la pluma está abajo, la tortuga traza las formas mientras se mueve; mientras la pluma está arriba, la tortuga se mueve libremente sin dibujar. En este problema, usted simulará la operación de la tortuga, así cómo el tablero computarizado.

Utilice un arreglo de 50 por 50 llamado piso**,** inicializado en ceros. Lea los comandos desde un arreglo que los contenga. Mantenga la pista de la posición actual de la tortuga en todo momento, y si la pluma está arriba o abajo. Suponga que la tortuga comienza siempre en la posición 0,0 del piso con la pluma arriba. El conjunto de los comandos de la tortuga que usted debe procesar aparece en la figura 6.24. Suponga que la tortuga se encuentra en algún lugar cerca del centro del piso. El siguiente “programa” debe dibujar y desplegar un cuadrado de 12 por 12:

```
2
5, 12
3
5, 12
3
5, 12
3
5, 12
1
6
9
```

Mientras la tortuga se mueva con la pluma abajo, establezca en **1**s los elementos apropiados del arreglo piso. Cuando se introduzca el comando **6** (imprimir), donde quiera que se encuentre un **1** dentro del arreglo, despliegue un asterisco o algún otro carácter que elija. Donde quiera que haya un cero, despliegue un blanco. Escriba un programa para implementar las capacidades de los gráficos de tortuga que explicamos aquí. Escriba varios programas de gráficos de tortuga para dibujar formas interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

**6.24** (*El recorrido del caballo.*) Uno de los juegos de intriga más interesantes para los entusiastas del ajedrez es el problema del Recorrido del caballo. La pregunta es: ¿puede una pieza de ajedrez llamada caballo moverse alrededor de un tablero y tocar cada una de las 64 posiciones, una y sólo una vez? Aquí estudiaremos este intrigante problema a fondo.

El caballo tiene un movimiento en forma de L (dos posiciones en una dirección y una posición en dirección perpendicular). Por lo tanto, a partir de un cuadrado en el centro de un tablero, el caballo puede hacer ocho movimientos diferentes (numerados de 0 a 7) como muestra la figura 6.25.

a) Dibuje un tablero de ajedrez de 8 por 8 en una hoja de papel e intente el recorrido del caballo a mano. Coloque un **1** en la primera posición a la que se mueva, un **2** en la segunda posición, un **3** en la tercera, etcétera. Antes de comenzar el recorrido, estime qué tan lejos cree usted que llegará, recuerde que el recorrido completo consiste en 64 movimientos. ¿Qué tan lejos llegó? ¿Fue lo que usted estimó?

| Comando | Significado |
|---|---|
| 1 | Pluma arriba |
| 2 | Pluma abajo |
| 3 | Vuelta a la derecha |
| 4 | Vuelta a la izquierda |
| 5, 10 | Movimiento hacia adelante 10 posiciones (u otro número diferente de 10) |
| 6 | Despliega el arreglo de 50 por 50 |
| 9 | Fin de datos (centinela) |

**Figura 6.24** Comandos de tortuga.

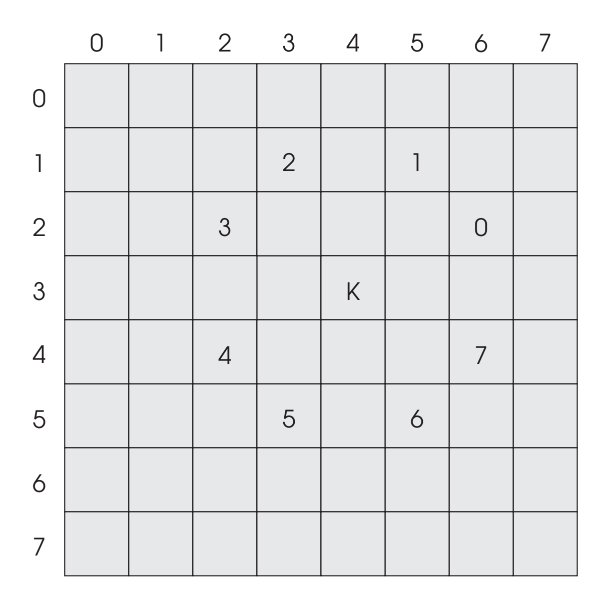



**Figura 6.25** Los ocho posibles movimientos del caballo.

b) Ahora, desarrollemos un programa que mueva el caballo alrededor del tablero. El tablero se representa mediante un arreglo con dos subíndices de 8 por 8, llamado tablero. Cada una de las posiciones se inicializa en cero. Describimos cada uno de los ocho posibles movimientos en términos tanto de su componente horizontal como de la vertical. Por ejemplo, un movimiento de tipo 0, como lo muestra la figura 6.25, consiste en moverse una posición a la izquierda y dos posiciones verticales hacia arriba. Los movimientos horizontales a la izquierda y los movimientos verticales hacia arriba se indican con números negativos. Los ochos movimientos se deben describir mediante dos arreglos con dos subíndices, **horizontal** y **vertical**, como sigue:

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

Haga que las variables **filaActual** y **columnaActual** indiquen la fila y la columna de la posición actual del caballo. Para hacer un movimiento de tipo **numeroMovimiento**, en donde **numeroMovimiento** se encuentra entre 0 y 7, su programa utiliza las instrucciones

```
filaActual += vertical[ numeroMovimiento ];
columnaActual += horizontal[ numeroMovimiento ];
```

Mantenga un contador que varíe de **1** a **64**. Registre la última cuenta en cada posición a la que el caballo se mueve. Recuerde probar cada movimiento posible para ver si el caballo ya ha visitado dicha posición, y, por supuesto, pruebe en el probable movimiento que el caballo no ha pisado fuera del tablero. Escriba ahora un programa para mover el caballo alrededor del tablero. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

c) Después de escribir y ejecutar el programa del recorrido del caballo, probablemente haya desarrollado sus propias ideas valiosas. Utilizaremos estas ideas para desarrollar una *heurística* (estrategia) para mover el caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora en gran medida la oportunidad de éxito. Probablemente usted ya observó que las posiciones externas son más difíciles que las posiciones cercanas al centro del tablero. De hecho, las posiciones más difíciles, o inaccesibles son las cuatro esquinas.
La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que es más fácil acceder, de manera que cuando el tablero se congestione cerca del final del recorrido, habrá una mayor oportunidad de éxito.
Debemos desarrollar una "heurística de accesibilidad", clasificando cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo a la posición (con los movimientos en L del caballo, por supuesto) que son más accesibles. Etiquetamos el arreglo con dos subíndices, **`accesibilidad`**, con los números que indican desde cuántas posiciones es accesible una posición determinada. Sobre un tablero en blanco, cada posición central tiene un grado de **8**, cada esquina tiene un grado **2** y las otras posiciones tienen números de accesibilidad **3**, **4** o **6** de la siguiente manera:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

Ahora, escriba una versión del programa del recorrido del caballo, utilizando la heurística de accesibilidad. El caballo se puede mover en cualquier momento a la posición con el número menor de accesibilidad. En caso de un empate, el caballo se puede mover a cualquiera de las posiciones con empate. Por lo tanto, el recorrido puede comenzar en cualquiera de las cuatro esquinas. (*Nota:* Mientras el caballo se mueve alrededor del tablero, su programa debe reducir los números de accesibilidad al ocuparse más y más posiciones. De esta manera, en cualquier momento durante el recorrido, cada número de posición disponible permanecerá igual al número preciso de posiciones a partir de la cual se puede acceder a dicha posición). Ejecute esta versión de su programa. ¿Obtuvo el recorrido completo? Modifique ahora el programa para ejecutar 64 recorridos, Y que cada uno comience en una posición del tablero. ¿Cuántas rutas completas obtuvo?

d) Escriba una versión del programa del recorrido del caballo, la cual, cuando encuentre un empate entre dos o más posiciones, decida cuál posición elegir, buscando aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su programa se debe mover a la posición en la que el siguiente movimiento alcance a la posición con el número menor de accesibilidad.

**6.25** *(Recorrido del caballo: métodos de fuerza bruta.)* En el ejercicio 6.24, desarrollamos una solución para el problema del recorrido del caballo. El método utilizado, llamado "*heurística de accesibilidad*", genera muchas soluciones y se ejecuta de manera eficiente.
Mientras se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y con algoritmos relativamente menos sofisticados. Llamemos a éste el método de la "fuerza bruta" para resolver un problema.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (por supuesto, mediante sus movimientos en L) de manera aleatoria. Su programa debe ejecutar un recorrido e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, el método anterior produce recorridos relativamente cortos. Ahora modifique su programa para intentar 1000 recorridos. Utilice un arreglo con un solo subíndice para dar seguimiento al número de recorridos de cada longitud. Cuando su programa termine los 1000 recorridos, deberá desplegar esta información en un ordenado formato tabular. ¿Cuál fue el mejor resultado?

c) Es muy probable que la mayoría de las veces, el programa anterior le haya brindado algunos recorridos "respetables", pero no recorridos completos. Ahora "suéltele la rienda" y simplemente deje que su programa se ejecute hasta que produzca un paso completo. (*Precaución:* Esta versión del programa podría ejecutarse durante horas en una computadora poderosa). Una vez más, mantenga una tabla con el número de recorridos para cada longitud, y despliegue esta tabla cuando se genere el primer recorrido completo. ¿Cuántos recorridos intentó su programa antes de generar un recorrido completo? ¿Cuánto tiempo se tomó?

d) Compare la versión de la fuerza bruta del recorrido del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más detallado del problema? ¿Cuál algoritmo fue más difícil de desarrollar? ¿Cuál requirió

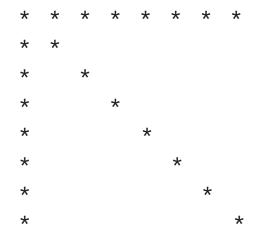


**Figura 6.26** Los 22 cuadros eliminados al colocar una reina en la esquina superior izquierda.

más potencia de la computadora? ¿Podríamos tener la certeza (por adelantado) de obtener un recorrido completo mediante el método de la fuerza bruta? Argumente las ventajas y las desventajas de solucionar el problema mediante la fuerza bruta en general.

**6.26** (*Ocho reinas.*) Otro enigma para los amantes del ajedrez es el problema de las ocho reinas, el cual dice: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna reina ataque a otra, es decir, que dos reinas no estén en la misma fila, en la misma columna, o a lo largo de la misma diagonal? Utilice la idea desarrollada en el ejercicio 6.24 para formular la heurística para resolver el problema de las ocho reinas. Ejecute su programa. [*Pista:* Es posible asignar un valor a cada cuadro del tablero, que indique cuántos cuadros de un tablero vacío son "eliminados" si se coloca una reina en ese cuadro. Por ejemplo, a cada una de las esquinas se le asignaría el valor 22, como en la figura 6.26.]

Una vez que estos "números de eliminación" se colocan en los 64 cuadros, una heurística adecuada podría ser: coloque la siguiente reina en el cuadro que tenga el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?

**6.27** (*Ocho reinas: métodos de fuerza bruta.*) En este ejercicio, usted desarrollará diversos métodos para resolver el problema de las ocho reinas que presentamos en el ejercicio 6.26.

a) Resuelva el ejercicio de las ocho reinas, utilizando la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 6.25.
b) Utilice una técnica exhaustiva, es decir, intente todas las posibles combinaciones de las ocho reinas en el tablero.
c) ¿Por qué supone que el método exhaustivo de la fuerza bruta puede no resultar apropiado para resolver el problema del recorrido del caballo?
d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva en general.

**6.28** (*Eliminación de duplicados.*) En el capítulo 12, se explora la estructura de datos árbol de búsqueda binaria de alta velocidad. Una característica del árbol de búsqueda binaria es que los valores duplicados se descartan cuando se hacen inserciones en el árbol. A esto se le conoce como eliminación de duplicados. Escriba un programa que produzca 20 números aleatorios entre 1 y 20. El programa debe almacenar en un arreglo todos los valores no duplicados. Utilice el arreglo más pequeño posible para llevar a cabo esta tarea.

**6.29** (*Recorrido del caballo: prueba del paseo cerrado.*) En el recorrido del caballo, ocurre un recorrido completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un recorrido cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de donde el caballo inició su paseo. Modifique el programa del recorrido del caballo que escribió en el ejercicio 6.24, para probar si el recorrido ha sido completo, y si se trató de un paseo cerrado.

**6.30** (*El cedazo de Eratóstenes.*) Un entero primo es cualquier entero divisible sólo por sí mismo y por el número 1. El método del cedazo de Eratóstenes se utiliza para localizar números primos. Éste funciona de la siguiente manera:

1) Crea un arreglo con todos los elementos inicializados en 1 (verdadero). Los elementos del arreglo con subíndices primos permanecerán como 1. Los demás elementos, en algún momento se establecerán en cero.
2) Comienza con un subíndice 2, cada vez que se encuentra un elemento del arreglo cuyo valor es 1, repite a lo largo del resto del arreglo y establece en cero cada elemento cuyo subíndice sea múltiplo del subíndice para el elemento con valor de 1. Para un subíndice 2 del arreglo, todos los elementos que pasen de 2 y que sean múltiplos de 2, se establecerán en cero (subíndices 4, 6, 8, 10, etcétera); para un subíndice de 3, todos los elementos que pasen de 3 y que sean múltiplos de 3, se establecerán en cero (subíndices 6, 9, 12, 15, etcétera).

Cuando este proceso termina, los elemento del arreglo que aún permanecen en 1, indican que el subíndice es un número primo. Estos subíndices pueden entonces desplegarse. Escriba un programa que utilice un arreglo de 1000 elementos para determinar y desplegar los números primos entre el 2 y el 999. Ignore el elemento 0 del arreglo.

**6.31** (*Ordenamiento por cubetas.*) Un ordenamiento por cubetas comienza con un arreglo de enteros positivos con un solo subíndice para ser ordenados, y un arreglo de enteros con dos subíndices, con filas cuyos subíndices se encuentran entre el 0 y el 9, y columnas cuyos subíndices van del 0 a $n-1$, en donde $n$ es el número de valores del arreglo a ordenarse. A cada fila del arreglo con dos subíndices se le conoce como cubeta. Escriba una función **`ordenamientoCubeta`** que tome como argumentos un arreglo entero y el tamaño del arreglo.

El algoritmo es el siguiente:

1) Hace un ciclo a través del arreglo con un solo subíndice y coloca cada uno de sus valores en una fila del arreglo en cubetas, basándose en los valores de uno de sus dígitos. Por ejemplo, el 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0.
2) Hace un ciclo a lo largo del arreglo en cubetas, fila por fila, y copia los valores nuevamente en el arreglo original. El nuevo orden de los valores anteriores, en el arreglo con un solo subíndice, es 100, 3 y 97.
3) Repite este proceso para cada posición subsiguiente de los dígitos (décimas, centésimas, milésimas, etcétera) y se detiene cuando el dígito que se encuentra más a la izquierda del número más grande se ha procesado.

En la segunda pasada, el 100 se coloca en la fila 0, el 3 en la fila 0 (ya que 3 no tiene décimas) y 97 se coloca en la fila 9. El orden de los valores del arreglo con un solo subíndice es 100, 3, 97. En la tercera pasada, 100 se coloca en la fila 1, el 3 en la fila cero y el 97 en la fila cero (después del 3). Se garantiza que el ordenamiento por cubetas tenga ordenados adecuadamente a todos los valores, después de procesar al dígito más a la izquierda del número más grande. El ordenamiento por cubetas sabe que esto está hecho, cuando todos los valores se copian en la fila cero del arreglo con dos subíndices.

Observe que el arreglo cubetas con dos subíndices tiene 10 veces el tamaño del arreglo entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que un ordenamiento de burbuja, pero requiere mucha más memoria. El ordenamiento de burbuja sólo requiere espacio para un elemento de datos adicional. El ordenamiento por cubetas es un ejemplo de la desventaja espacio-tiempo éste utiliza más memoria, pero se desempeña mejor. Esta versión del ordenamiento por cubetas requiere que se copien todos los datos nuevamente en el arreglo original en cada paso. Otra posibilidad es crear un segundo arreglo con dos subíndices, y repetidamente mover los datos entre los dos arreglos cubetas, hasta que los datos se copien en la fila cero de uno de los arreglos. La fila cero entonces contiene el arreglo ordenado.

## EJERCICIOS DE RECURSIVIDAD

**6.32** (*Ordenamiento por selección.*) Un ordenamiento por selección busca un arreglo que busca al elemento más pequeño del arreglo. Después, el elemento más pequeño se intercambia por el primer elemento del arreglo. El proceso se repite para el subarreglo, comenzando con el segundo elemento del arreglo. Cada pasada en el arreglo da como resultado a un elemento que se coloca en su propia ubicación. Este ordenamiento se desempeña de manera similar al ordenamiento de burbuja; para un arreglo de $n$ elementos, es necesario realizar $n-1$ pasos, y para cada subarreglo deben hacerse $n-1$ comparaciones para encontrar el valor más pequeño. Cuando el subarreglo que se está procesando contiene un elemento, el arreglo esta ordenado. Escriba la función recursiva **`ordenamientoSeleccion`**, para desarrollar este algoritmo.

**6.33** (*Palíndromos.*) Un palíndromo es una cadena que dice lo mismo si se lee hacia delante que si se lee hacia atrás. Algunos ejemplos de palíndromos son “radar”, “ojo”, “oso”. Escriba una función recursiva **`pruebaPalindromo`** que devuelva 1 si la cadena almacenada en el arreglo es un palíndromo, y 0 si no lo es. La función debe ignorar los espacios y la puntuación en la cadena.

**6.34** (*Búsqueda lineal.*) Modifique el programa de la figura 6.18 para utilizar la función recursiva **`busquedaLineal`** para realizar una búsqueda lineal en el arreglo. La función debe recibir un arreglo entero y el tamaño del arreglo como sus argumentos. Si la clave de búsqueda se localiza, devuelva el subíndice del arreglo; de otro modo devuelva $-1$.

**6.35** (*Búsqueda binaria.*) Modifique el programa de la figura 6.19 para utilizar una función recursiva **`busquedaBinaria`**, para realizar la búsqueda binaria en el arreglo. La función debe recibir un arreglo entero y el subíndice inicial y el final como sus argumentos. Si la clave de búsqueda es localizada, devuelva el subíndice del arreglo; de otro modo devuelva $-1$.

**6.36** (*Ocho reinas.*) Modifique el programa de las ocho reinas que creó en el ejercicio 6.26, para resolver el problema de manera recursiva.

**6.37** (*Impresión de un arreglo.*) Escriba una función recursiva **`desplegarArreglo`** que tome un arreglo y el tamaño del arreglo como sus argumentos y que no devuelva valor alguno. La función debe detener el procesamiento y regresar, cuando reciba un arreglo de tamaño cero.

**6.38** (*Impresión de una cadena al revés.*) Escriba una función recursiva **`cadenaAlReves`**, que tome un arreglo de caracteres que contenga una cadena como un argumento, que despliegue la cadena al revés y que no devuelva valor alguno. La función debe detener el procesamiento y regresar, cuando encuentre el carácter de terminación nulo.

**6.39** (*Cómo encontrar el valor mínimo de un arreglo.*) Escriba una función recursiva **`minimoRecursivo`**, que tome un arreglo entero y el tamaño del arreglo como argumentos y que devuelva el elemento más pequeño del arreglo. La función debe detener el procesamiento y regresar, cuando reciba un arreglo de un elemento.

# 7

# Apuntadores en C

## Objetivos

- Comprender los apuntadores y los operadores para apuntadores.
- Utilizar los apuntadores para pasar por referencia argumentos a una función.
- Comprender las relaciones entre apuntadores, arreglos y cadenas.
- Comprender el uso de los apuntadores a funciones.
- Definir y utilizar los arreglos de cadenas.




*Las direcciones se nos dan para ocultar nuestro paradero.*
Saki (H. H. Munro)

*Mediante rodeos encuentra el rumbo.*
William Shakespeare
*Hamlet*

*Muchas cosas, conociéndolas bien,*
*con el consentimiento de uno, pueden funcionar de manera contraria.*
William Shakespeare
*King Henry V*

*¡Usted descubrirá que una buena práctica es siempre verificar sus referencias!*
Dr. Routh

*Usted no puede confiar en código que no genere usted completamente.*
*(Especialmente en código de empresas que emplean a gente como yo.)*
Ken Thompson
Turin Award Lecture, 1983

## Plan general



## 7.1 Introducción

En este capítulo, explicaremos una de las características más poderosas del lenguaje de programación C, el *apuntador*. Los apuntadores son de las capacidades de C más difíciles de dominar. Los apuntadores permiten a los programadores simular las llamadas por referencia, y crear y manipular estructuras de datos dinámicas, es decir, estructuras de datos que pueden crecer y encogerse en tiempo de ejecución, tales como *listas ligadas*, colas, pilas y árboles. En este capítulo, explicamos los conceptos básicos de los apuntadores. En el capítulo 10 explicaremos cómo utilizar los apuntadores con estructuras. En el capítulo 12 introducimos las técnicas de *administración de memoria dinámica* y presentamos ejemplos para la creación y el uso de estructuras de datos dinámicas.

## 7.2 Definición e inicialización de variables de apuntador

Los apuntadores son variables cuyos valores son direcciones de memoria. Por lo general, una variable contiene directamente un valor específico. Por otro lado, un apuntador contiene la dirección de una variable que contiene un valor específico. En este sentido, el nombre de una variable hace referencia *directa* a un valor, y un apuntador hace referencia *indirecta* a un valor (figura 7.1). Al proceso de referenciar a un valor a través de un apuntador se le llama *indirección*.

Los apuntadores, como todas las variables, deben definirse antes de que se puedan utilizar. La definición

```
int *ptrCuenta, cuenta;
```

especifica que la variable **ptrCuenta** es de tipo **int *** (es decir, un apuntador a un entero) y se lee, "**ptrCuenta** es un apuntador a un **int**" o "**ptrCuenta** apunta a un objeto de tipo **int**". Además, la variable **cuenta** se define como **int**, no como un apuntador a un **int**. El ***** sólo se aplica a la variable que se define como apuntador. Cuando se utiliza el ***** de este modo en una definición, indica que la variable que se está definiendo es un apuntador. Los apuntadores pueden definirse para apuntar a objetos de cualquier tipo de dato.



**Error común de programación 7.1**

*La notación asterisco (*****) que se utiliza para declarar variables de tipo apuntador no se distribuye a todas las variables en la declaración. Cada apuntador debe declararse con el prefijo ***** en el nombre, por ejemplo, si desea declarar **ptrX** y **ptrY** como apuntadores int, utilice **int *ptrX, *ptrY**;*

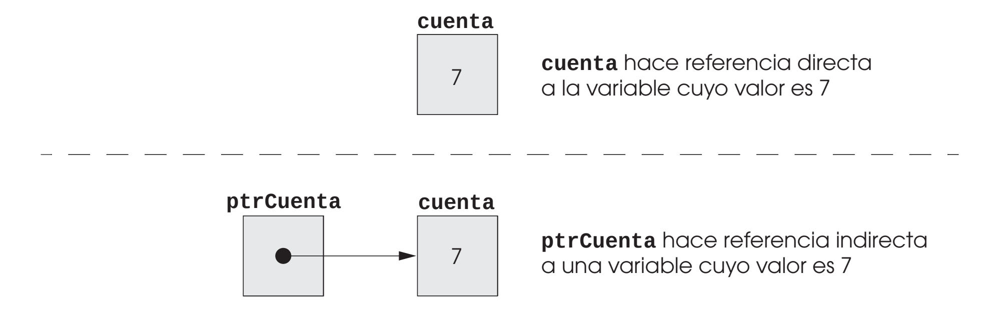



**Figura 7.1** Referencias directa e indirecta a una variable.



**Buena práctica de programación 7.1**

*Incluya las letras* **ptr** *en los nombres de las variables de apuntadores para hacer más claro que estas variables son apuntadores y, por lo tanto, que deben manipularse de manera apropiada.*

Los apuntadores deben inicializarse en el momento en que se definen o en una instrucción de asignación. Un apuntador puede inicializarse en **0**, **NULL** o en una dirección. Un apuntador con el valor **NULL**, apunta a nada. **NULL** es una constante simbólica definida en el encabezado **<stddef.h>** (el cual se incluye en varios otros encabezados, tales como **<stdio.h>**). Inicializar un apuntador en **0** es equivalente a inicializar un apuntador en **NULL**, pero es preferible utilizar **NULL**. Cuando se asigna **0**, primero se convierte a un apuntador del tipo apropiado. El valor **0** es el único valor entero que puede asignarse de manera directa a la variable de apuntador. En la sección 7.3 explicaremos la asignación de la dirección de una variable a un apuntador.



**Tip para prevenir errores 7.1**

*Inicialice los apuntadores para prevenir resultados inesperados.*

## 7.3 Operadores para apuntadores

El &, u *operador de dirección*, es un operador unario que devuelve la dirección de su operando. Por ejemplo, si consideramos las definiciones

```
int y = 5;
int *ptrY;
```

la instrucción

```
ptrY = &y;
```

asigna la dirección de la variable y a la variable apuntador **ptrY**. Entonces, se dice que la variable **ptrY** "apunta a" **y**. En la figura 7.2 mostramos una representación esquemática de la memoria, después de que se ejecuta la instrucción anterior.

La figura 7.3 muestra la representación del apuntador en memoria, asumiendo que la variable entera **y** está almacenada en la dirección de memoria **600000**, y que la variable de apuntador **ptrY** está almacenada en la ubicación de memoria **500000**. El operando del operador de dirección debe ser una variable; el operador de

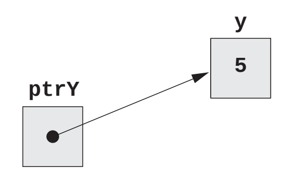



**Figura 7.2** Representación gráfica de un apuntador que apunta hacia una variable entera en memoria.




**Figura 7.3** Representación en memoria de **`y`** y **`ptrY`**.

dirección no puede aplicarse a constantes, expresiones o variables declaradas mediante la clase de almacenamiento **`register`**.

El operador *, por lo general llamado *operador de indirección* u *operador de desreferencia*, devuelve el valor del objeto al que apunta su operando (es decir, un apuntador). Por ejemplo, la instrucción

```
printf( "%d", *ptrY );
```

imprime el valor de la variable **`y`**, a saber 5. Al uso de * de esta manera se le conoce como *desreferenciar a un apuntador*.



### Error común de programación 7.2

*Desreferenciar un apuntador que no se inicializó de manera apropiada, o que no se le indicó que apunte hacia una dirección específica en memoria es un error. Esto podría provocar un error fatal en tiempo de ejecución, o podría modificar de manera accidental datos importantes y permitir la ejecución del programa pero con resultados incorrectos.*

La figura 7.4 muestra los operadores **`&`** y **`*`**. En la mayoría de las plataformas, el especificador de conversión de **`printf`**, **`%p`**, despliega la ubicación en memoria como un entero hexadecimal. (Vea el apéndice E, Sistemas de Numeración, para mayor información acerca de los enteros hexadecimales.) Observe que la dirección de **`a`** y el valor de **`ptrA`** son idénticos en la salida, esto confirma que la dirección de **`a`** realmente se asigna a la variable apuntador **`ptrA`** (línea 11). Los operadores **`&`** y **`*`** son complementos uno del otro, cuando ambos se aplican de manera consecutiva a **`ptrA`**, en cualquier orden (línea 21), se imprime el mismo resultado. La figura 7.5 lista la precedencia y asociatividad de los operadores que hemos presentado hasta este punto.

```
1  /* Figura 7.4: fig07_04.c
2     Uso de los operadores & y * */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;         /* a es un entero */
8     int *ptrA;     /* ptrA es un apuntador a un entero */
9
10    a = 7;
11    ptrA = &a;     /* ptrA toma la dirección de a */
12
13    printf( "La direccion de a es %p"
14            "\nEl valor de ptrA es %p", &a, ptrA );
15
16    printf( "\n\nEl valor de a es %d"
17            "\nEl valor de *ptrA es %d", a, *ptrA );
18
19    printf( "\n\nMuestra de que * y & son complementos "
20            "uno del otro\n&*ptrA = %p"
21            "\n*&ptrA = %p\n", &*ptrA, *&ptrA );
22
23    return 0; /* indica terminación exitosa */
```

**Figura 7.4** Operadores & y * con apuntadores. (Parte 1 de 2.)

```

} /* fin de main */
```

```
La direccion de a es 0012FF7C
El valor de ptrA es 0012FF7C

El valor de a es 7
El valor de *ptrA es 7

Muestra de que * y & son complementos uno del otro
&*ptrA = 0012FF7C
*&ptrA = 0012FF7C
```

**Figura 7.4** Operadores & y * con apuntadores. (Parte 2 de 2.)

| Operadores | Asociatividad | Tipo |
|---|---|---|
| `()  []` | izquierda a derecha | más alto |
| `+  -  ++  --  !  *  &  (tipo)` | derecha a izquierda | unario |
| `*  /  %` | izquierda a derecha | de multiplicación |
| `+  -` | izquierda a derecha | de suma |
| `<  <=  >  >=` | izquierda a derecha | de relación |
| `==  !=` | izquierda a derecha | de igualdad |
| `&&` | izquierda a derecha | and lógico |
| `\|\|` | izquierda a derecha | or lógico |
| `?:` | derecha a izquierda | condicional |
| `=  +  =  -=  *=  /=  %=` | derecha a izquierda | de asignación |
| `,` | izquierda a derecha | coma |

**Figura 7.5** Precedencia de operadores.

## 7.4 Llamada a funciones por referencia

Existen dos maneras de pasar argumentos a una función: mediante *llamadas por valor* y mediante *llamadas por referencia*. Todos los argumentos de C se pasan por valor. Como vimos en el capítulo 5, **`return`** puede utilizarse para devolver un valor desde la función invocada hacia la llamada de la función (o para devolver el control desde una función invocada, sin devolver valor alguno). Muchas funciones requieren la capacidad de modificar una o más variables en la llamada de la función, o pasar un apuntador a un objeto grande para evitar la sobrecarga de pasar objetos por valor (lo que provoca la sobrecarga de hacer copias del objeto). Para estos propósitos, C proporciona las capacidades para *simular las llamadas por referencia*.

En C, los programadores utilizan apuntadores y el operador de indirección para simular las llamadas por referencia. Cuando llamamos a una función con argumentos que deben modificarse, se pasan las direcciones de dichos argumentos. Por lo general esto se lleva a cabo mediante la aplicación (en la llamada a la función) del operador de dirección (**&**) a la variable cuyo valor se modificará. Como vimos en el capítulo 6, los arreglos no se pasan mediante el operador **&** debido a que C pasa de manera automática la dirección inicial en memoria del arreglo (el nombre del arreglo es equivalente a **`&nombreArreglo[ 0 ]`**). Cuando la dirección de una variable se pasa a una función, debemos utilizar el operador de indirección (**`*`**) en la función, para modificar el valor de dicha ubicación en la memoria de la llamada a la función.

Los programas de las figuras 7.6 y 7.7 presentan dos versiones de una función que eleva al cubo un entero, **`cuboPorValor`** y **`cuboPorReferencia`**. La figura 7.6 pasa la variable **`numero`** a la función **`cuboPorValor`** mediante una llamada por valor (línea 14). La función **`cuboPorValor`** eleva al cubo su argu-

```
/* Figura 7.6: fig07_06.c
   Eleva al cubo una variable mediante una llamada por valor */
#include <stdio.h>

int cuboPorValor( int n ); /* prototipo */

int main()
{
   int numero = 5; /* inicializa numero */

   printf( "El valor original de numero es %d", numero );

   /* pasa numero por valor a cuboPorValor */
   numero = cuboPorValor( numero );

 printf( "\nEl nuevo valor de numero es %d\n", numero );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* calcula y devuelve el cubo de un argumento entero */
int cuboPorValor( int n )
{
   return n * n * n;   /* eleva al cubo la variable local n y devuelve
   el resultado */

} /* fin de la función cuboPorValor */
```

```
El valor original de numero es 5
El nuevo valor de numero es 125
```

**Figura 7.6** Cómo elevar al cubo una variable mediante una llamada por valor.

```
/* Figura 7.7: fig07_07.c
   Eleva al cubo una variable mediante una llamada por referencia, con un
   apuntador como argumento */

#include <stdio.h>

void cuboPorReferencia( int *ptrN ); /* prototipo */

int main()
{
   int numero = 5; /* inicializa numero */

   printf( "El valor original de numero es %d", numero );

   /* pasa la dirección de numero a cuboPorReferencia */
   cuboPorReferencia( &numero );

   printf( "\nEl nuevo valor de numero es %d\n", numero );

   return 0; /* indica terminación exitosa */
```

**Figura 7.7** Cómo elevar al cubo una variable mediante una llamada por referencia. (Parte 1 de 2.)

```
20
21 } /* fin de main */
22
23 /* calcula el cubo de *ptrN; modifica la variable numero en main */
24 void cuboPorReferencia( int *ptrN )
25 {
26    *ptrN = *ptrN * *ptrN * *ptrN;  /* cubo de *ptrN */
27 } /* fin de la función cuboPorReferencia */
```

```
El valor original de numero es 5
El nuevo valor de numero es 125
```

**Figura 7.7** Cómo elevar al cubo una variable mediante una llamada por referencia. (Parte 2 de 2.)

mento y pasa de regreso el nuevo valor a **main** mediante la instrucción **return**. El nuevo valor se asigna a **numero** en **main** (línea 14).

La figura 7.7 pasa la variable **numero** mediante una llamada por referencia (línea 15); se pasa la dirección de **numero** a la función **cuboPorReferencia**. La función **cuboPorReferencia** toma como parámetro un apuntador hacia un **int** llamado **ptrN** (línea 24). La función desreferencia al apuntador y eleva al cubo el valor al cual apunta **ptrN** (línea 26), después asigna el resultado a ***ptrN** (que es en realidad **numero** en **main**), y así, modifica el valor de **numero** en **main**. Las figuras 7.8 y 7.9 analizan de manera gráfica los programas de las figuras 7.6 y 7.7, respectivamente.

Antes de que **main** llame a **cuboPorValor**:

```
int main()                          numero
{                                   [ 5 ]
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

```
int cuboPorValor( int n )
{
   return n * n * n;
}
                         n
                  [ indefinido ]
```

Después de que **main** llama a **cuboPorValor**:

```
int main()                          numero
{                                   [ 5 ]
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

```
int cuboPorValor( int n )
{
   return n * n * n;
}
                         n
                       [ 5 ]
```

Después del parámetro n de **cuboPorValor** y antes de que **cuboPorValor** retorne a **main**:

```
int main()                          numero
{                                   [ 5 ]
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

```
int cuboPorValor( int n )
{           125
   return n * n * n;
}
                         n
                       [ 5 ]
```

**Figura 7.8** Análisis de una típica llamada por valor. (Parte 1 de 2.)

Después de que **cuboPorValor** retorna a **main** y antes de que asigne el resultado a numero:

```
int main()                          numero
{    int numero = 5;                  5
                          125
   numero = cuboPorValor( numero );
}
```

```
int cuboPorValor( int n )
{
   return n * n * n;
}                n
             indefinido
```

Después de que **main** completa la asignación a numero:

```
int main()                          numero
{                                    125
int numero = 5;
  125             125
   numero = cuboPorValor( numero );
}
```

```
int cuboPorValor( int n )
{
   return n * n * n;
}                n
             indefinido
```

**Figura 7.8** Análisis de una típica llamada por valor. (Parte 2 de 2.)

## Error común de programación 7.3

*No desreferenciar un apuntador cuando es necesario hacerlo para obtener el valor al que apunta el apuntador, es un error de sintaxis.*

Una función que recibe como argumento una dirección, debe definir un parámetro de apuntador para recibir la dirección. Por ejemplo, en la figura 7.7 el encabezado de la función **cuboPorReferencia** (línea 24) es:

```
void cuboPorReferencia( int *ptrN )
```

Antes de que **main** llame a **cuboPorReferencia**:

```
int main()                  numero
{                             5
   int numero = 5;

   cuboPorReferencia ( &numero );
}
```

```
void cuboPorReferencia( int *ptrN )
{
   *ptrN = *ptrN * *ptrN * *ptrN;
}
                          ptrN
                       indefinido
```

Después de que **cuboPorReferencia** recibe la llamada y antes de que ***ptrN** se eleve al cubo:

```
int main()                  numero
{                             5
   int numero = 5;

cuboPorReferencia ( &numero );
}
```

```
void cuboPorReferencia ( int *ptrN )
{
   *ptrN = *ptrN * *ptrN * *ptrN;
}
                                     ptrN
la llamada establece este apuntador    •
```

Después de que *__ptrN__ se eleva al cubo y antes de que el control del programa retorne a **main**:

```
int main()                  numero
{                            125
   int numero = 5;

   cuboPorReferencia ( &numero );
}
```

```
void cuboPorReferencia ( int *ptrN )
{                       125
*ptrN = *ptrN * *ptrN * *ptrN;
}
la función llamada modifica la       ptrN
variable de la función que llama       •
```

**Figura 7.9** Análisis de una típica llamada por referencia con un apuntador como argumento.

El encabezado especifica que **cuboPorReferencia** recibe como argumento la dirección de una variable entera, almacena la dirección de manera local en **ptrN** y no devuelve valor alguno.

El prototipo de función para **cuboPorReferencia** contiene **int *** entre paréntesis. Como con otros tipos de variables, no es necesario incluir los nombres de los apuntadores en el prototipo de la función. El compilador de C ignora los nombres que se incluyen con fines de documentación.

En el encabezado de la función y en el prototipo para una función que espera un arreglo de un solo subíndice como argumento, se puede utilizar la notación de apuntadores en la lista de parámetros de **cuboPorReferencia**. El compilador no diferencia entre una función que recibe un apuntador y una función que recibe un arreglo de un solo subíndice. Esto, por supuesto, significa que la función debe "saber" cuándo recibe un arreglo o simplemente una variable para la cual hace la llamada por referencia. Cuando el compilador encuentra un parámetro de función para un arreglo de un solo subíndice de la forma **int b[]**, el compilador convierte el parámetro a la notación de apuntadores **int *b**. Estas dos formas son intercambiables.



**Tip para prevenir errores 7.2**

*Utilice llamadas por valor para pasar argumentos a una función, a menos que la función que hace la llamada requiera explícitamente que la función que se invoca modifique el valor del argumento en el entorno de la función que hace la llamada. Esto previene modificaciones accidentales de los argumentos en la llamada de la función, y es otro ejemplo del principio del menor privilegio.*

## 7.5 Uso del calificador `const` con apuntadores

El calificador **const** permite a los programadores informar al compilador que no se debe modificar el valor particular de una variable. El calificador **const** no existía en las primeras versiones de C; el comité ANSI de C lo adicionó al lenguaje.



**Observación de ingeniería de software 7.1**

*El calificador **const** puede utilizarse para reforzar el principio del menor privilegio. Utilizar el principio del menor privilegio para diseñar software de manera apropiada, reduce el tiempo de depuración y los efectos colaterales indeseados, lo que hace a un programa más fácil de modificar y de mantener.*

**Tip de portabilidad 7.1**

*Aunque **const** está bien definido en el ANSI C, algunos compiladores no lo soportan.*

Gran cantidad de código heredado que se utilizó con las primeras versiones de C no utiliza **const** debido a que no estaba disponible. Por esta razón, existen muchas oportunidades de mejorar la ingeniería de software del viejo código en C.

Existen seis posibilidades para utilizar (o no) **const** con parámetros de funciones: dos mediante el paso de parámetros por valor y cuatro mediante el paso de parámetros por referencia. ¿Cómo elegir una de las seis posibilidades? Deje que el *principio del menor privilegio* sea su guía. Otorgue siempre espacio suficiente para que los datos y sus parámetros realicen una tarea específica, pero no más.

En el capítulo 5, explicamos que todas las llamadas en C son por valor, es decir, que se crea una copia del argumento en la llamada de la función y se pasa a la misma función. Si la copia se modifica en la función, el valor original en la llamada no cambia. En muchos casos, un valor que se pasa a una función se modifica para que la función pueda llevar a cabo su tarea. Sin embargo, en algunas instancias, el valor no debe alterarse en la llamada a la función, aun cuando solamente manipule una copia del valor original.

Considere una función que toma como argumentos un arreglo de un solo subíndice y su tamaño, e imprime el arreglo. Tal función debe repetir un ciclo a lo largo del arreglo y desplegar cada elemento del arreglo de manera individual. El tamaño del arreglo se utiliza en el cuerpo de la función para determinar el subíndice más alto del arreglo, de manera que el ciclo pueda terminar cuando se complete la impresión. Ni el tamaño del arreglo ni su contenido deben cambiar en el cuerpo de la función.



**Tip para prevenir errores 7.3**

*Si una variable no se modifica (o no debiera modificarse) en el cuerpo de la función a la que se pasa, la variable debe declararse como **const** para garantizar que no se modifique de manera accidental.*

Si se hace un intento de modificar un valor que se declara como **const**, el compilador lo atrapa y lanza un mensaje de error o de advertencia, dependiendo del compilador.



### Observación de ingeniería de software 7.2

*Sólo se puede alterar un valor en la función invocada cuado utilizamos una llamada por referencia. El valor debe asignarse desde el valor de retorno de la función. Para modificar valores en la función invocada, debe utilizar una llamada por referencia.*



### Tip para prevenir errores 7.4

*Antes de usar una función, verifique su prototipo para determinar si la función es capaz de modificar los valores que se le pasan.*

### Error común de programación 7.4

*No estar consciente de que una función espera apuntadores como argumentos para realizar una llamada por referencia y para pasar argumentos por valor. Algunos compiladores toman los valores y asumen que son apuntadores, por lo que desreferencian los valores como apuntadores. A tiempo de ejecución, a menudo generan violaciones de acceso a memoria o fallas de segmentación. Otros compiladores atrapan el error de tipos entre los argumentos y los parámetros, y generan mensajes de error.*

Existen cuatro formas de pasar un apuntador a una función: *un apuntador no constante a un dato no constante; un apuntador constante a un dato no constante; un apuntador no constate a un dato constante*, y *un apuntador constante a un dato constante*. Cada una de las cuatro combinaciones proporciona diferentes privilegios de acceso, los cuales explicaremos en los siguientes ejemplos.

***Cómo convertir una cadena a mayúsculas por medio de un apuntador no constante a un dato no constante***

El nivel más alto de acceso a datos lo brinda un apuntador no constante a un dato no constante. En este caso, los datos pueden modificarse a través de la desreferencia del apuntador, y el apuntador puede modificarse para que apunte a otros elementos. La declaración de un apuntador no constante a un dato no constante no incluye **const**. Se debe utilizar dicho apuntador para recibir una cadena como argumento de una función que utilice la *aritmética de apuntadores* para procesar (y posiblemente modificar) cada carácter de la cadena. La función **convierteAMayusculas** de la figura 7.10 declara su parámetro como un apuntador no constante a un dato no constante llamado **ptrS** (**char *ptrS** línea 23). La función procesa el arreglo **cadena** (al que apunta **ptrS**), carácter por carácter, mediante la aritmética de apuntadores. La función **islower** de la biblioteca estándar (llamada en la línea 27) verifica el contenido de la dirección a la que apunta **ptrS**. Si el carácter se encuentra en el rango de **'a'** a **'z'**, **islower** devuelve verdadero y se invoca a la función **toupper** de la biblioteca estándar (línea 28) para convertir el carácter a su letra correspondiente en mayúscula; de lo contrario, **islower** devuelve falso y se procesa el siguiente carácter de la cadena.

```
1  /* Figura 7.10: fig07_10.c
2     Conversión de letras minúsculas a letras mayúsculas
3     mediante un apuntador no constante a un dato no constante */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convierteAMayusculas( char *ptrS ); /* prototipo */
9
10 int main()
11 {
12    char cadena[] = "caracteres y $32.98"; /* inicializa char arreglo */
13
14    printf( "La cadena antes de la conversion es : %s", cadena );
15    convierteAMayusculas( cadena );
```

**Figura 7.10** Conversión de una cadena a mayúsculas por medio de un apuntador no constante a un dato no constante. (Parte 1 de 2.)

```
   printf( "\nLa cadena despues de la conversion es: %s\n", cadena );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* convierte una cadena a letras mayúsculas */
void convierteAmayusculas( char *ptrS )
{
   while ( *ptrS != '\0' ) { /* el carácter actual no es '\0' */

      if ( islower( *ptrS ) ) {     /* si el carácter es minúscula, */
         *ptrS = toupper( *ptrS ); /* Lo convierte a mayúscula */
      } /* fin de if */

      ++ptrS;  /* mueve ptrS al siguiente carácter */
   } /* fin del while */

} /* fin de la función convierteAMayusculas */
```

```
La cadena antes de la conversion es: caracteres y $32.98
La cadena despues de la conversion es: CARACTERES Y $32.98
```

**Figura 7.10** Conversión de una cadena a mayúsculas por medio de un apuntador no constante a un dato no constante. (Parte 2 de 2.)

***Cómo imprimir una cadena, carácter por carácter, mediante un apuntador no constante a un dato constante***

Podemos modificar un apuntador no constante a un dato constante para que apunte a cualquier elemento del tipo apropiado, pero no puede modificarse el dato al cual apunta. Dicho apuntador debe utilizarse para recibir un argumento de tipo arreglo para una función que procesará cada elemento del arreglo sin modificar los datos. Por ejemplo, la función **`imprimeCaracteres`** de la figura 7.11 declara el parámetro **`ptrS`** con el tipo **`const char *`** (línea 24). La declaración se lee de derecha a izquierda como "**`ptrS`** es un apuntador a una constante de carácter". El cuerpo de la función utiliza una instrucción **`for`** para mostrar cada carácter de la cadena hasta encontrar el carácter nulo.

```
1  /* Figura 7.11: fig07_11.c
2     Impresión de una cadena carácter por carácter mediante
3     un apuntador no constante a un dato constante */
4
5  #include <stdio.h>
6
7  void imprimeCaracteres( const char *ptrS );
8
9  int main()
10 {
11    /* inicializa el arreglo de caracteres */
12    char cadena[] = "imprime los caracteres de una cadena";
13
14    printf( "La cadena es:\n" );
15    imprimeCaracteres( cadena );
16    printf( "\n" );
17
```

**Figura 7.11** Impresión de una cadena carácter por carácter mediante un apuntador no constante a un dato constante. (Parte 1 de 2.)

```
   return 0; /* indica terminación exitosa */

} /* fin de main */

/* ptrS no puede modificar el carácter al cual apunta,
   es decir, ptrS es un apuntador de "solo lectura" */
void imprimeCaracteres( const char *ptrS )
{
   /* repite el ciclo para toda la cadena */
   for ( ; *ptrS != '\0'; ptrS++ ) {   /* sin inicialización */
      printf( "%c", *ptrS );
   } /* fin de for */

} /* fin de la función imprimeCaracteres */
```

```
La cadena es:
imprime los caracteres de una cadena
```

**Figura 7.11** Impresión de una cadena carácter por carácter mediante un apuntador no constante a un dato constante. (Parte 2 de 2.)

Después de la impresión de cada carácter, el apuntador **ptrS** se incrementa para que apunte al siguiente carácter de la cadena.

La figura 7.12 muestra los mensajes de error que emite el compilador al intentar compilar una función que recibe un apuntador no constante (**ptrX**) a un dato constante. Esta función intenta modificar el dato al que apunta **ptrX** en la línea 22, el cual provoca un mensaje de error. [*Nota:* El mensaje de error real que usted verá dependerá de cada compilador.]

```
1  /* Figura 7.12: fig07_12.c
2     Intenta modificar un dato a través de
3     un apuntador no constante a un dato constante. */
4  #include <stdio.h>
5
6  void f( const int *ptrX ); /* prototipo */
7
8  int main()
9  {
10    int y;        /* define y */
11
12    f( &y );      /* f intenta una modificación ilegal */
13
14    return 0;     /* indica terminación exitosa */
15
16 } /* fin de main */
17
18 /* no se puede utilizar ptrX para modificar
19    el valor de la variable a la cual apunta */
20 void f( const int *ptrX )
21 {
22    *ptrX = 100;  /* error: no se puede modificar un objeto const */
23 } /* fin de la función f*/
```

**Figura 7.12** Se intenta modificar los datos mediante un apuntador no constate a un dato constante. (Parte 1 de 2.)

```
Compiling...
fig07_12.c
c:\documents and settings\laura\configuración local\temp\fig07_12.c(22) : error
C2166: l-value specifies const object
Error executing cl.exe.

fig07_12.exe - 1 error(s), 0 warning(s)
```

**Figura 7.12** Se intenta modificar los datos mediante un apuntador no constate a un dato constante. (Parte 2 de 2.)

Como sabemos, los arreglos son tipos de datos agregados que almacenan elementos de datos relacionados, del mismo tipo y con el mismo nombre. En el capítulo 10, explicaremos otra forma de tipos de datos agregados llamados *estructuras* (y algunas veces en otros lenguajes llamados *registros*). Una estructura es capaz de almacenar datos relacionados de diferentes tipos con el mismo nombre (por ejemplo, almacena la información acerca del empleado de una empresa). Cuando se llama a una función que tiene un arreglo como argumento, el arreglo se pasa automáticamente por referencia a la función. Sin embargo, las estructuras siempre se pasan por valor; se pasa una copia completa de la estructura. Esto requiere la sobrecarga en tiempo de ejecución para hacer una copia de cada elemento de la estructura y para almacenarlo en la pila de llamadas a la función. Cuando la estructura de datos debe pasarse a una función, podemos utilizar apuntadores a datos constantes para obtener el rendimiento de la llamada por referencia y la protección de la llamada por valor. Cuando se pasa un apuntador a una estructura, sólo se hace una copia de la dirección en donde se almacena la estructura. En una máquina con direcciones de 4 bytes, se hace una copia de 4 bytes de memoria, en lugar de hacer una copia completa de los posibles cientos o miles de bytes de la estructura.

**Tip de rendimiento 7.1**

*El paso de objetos grandes, tales como estructuras, utilizando apuntadores a datos constantes, obtiene las ventajas de una llamada por referencia y la seguridad de una llamada por valor.*

Utilizar apuntadores a datos constantes de esta manera es un ejemplo del *equilibrio tiempo/espacio*. Si la memoria es poca y la eficiencia de la ejecución es una preocupación mayor, debe utilizar apuntadores. Si la memoria es abundante y la eficiencia no es una preocupación mayor, los datos se deben pasar por valor para promover el principio del menor privilegio. Recuerde que algunos sistemas no promueven bien el uso de `const`, de modo que la llamada por valor es la mejor manera de prevenir la modificación de los datos.

***Intento de modificar un apuntador constante a un dato no constante***

Un apuntador constante a un dato no constante siempre apunta a la misma ubicación de memoria, y el dato en esa ubicación puede modificarse a través del apuntador. Esto se da de manera predeterminada para el nombre de un arreglo. Un nombre de arreglo es un apuntador constante hacia el principio del arreglo. Se puede acceder a todos los datos del arreglo y modificarse mediante el nombre del arreglo y sus subíndices. Podemos utilizar un apuntador constante a un dato no constante para recibir un arreglo como argumento de la función que accede a los elementos del arreglo mediante la notación de subíndices. Los apuntadores que se declaran `const` deben inicializarse al momento de definirse (si el apuntador es un parámetro de función, se inicializa mediante el apuntador que se pasa a la función). El programa de la figura 7.13 intenta modificar un apuntador constante. El

```
/* Figura 7.13: fig07_13.c
   Intenta modificar un apuntador constante a un dato no constante */
#include <stdio.h>

int main()
{
   int x; /* define x */
```

**Figura 7.13** Intento de modificar un apuntador constante a un dato no constante. (Parte 1 de 2.)

```
8      int y; /* define y */
9
10     /* ptr es un apuntador constante a un entero que se puede modificar
11        a través de ptr, pero ptr siempre apunta a la misma ubicación
          de memoria */
12     int * const ptr = &x;
13
14     *ptr = 7; /* permitido: *ptr no es const */
15     ptr = &y; /* error: ptr es const; no se puede asignar una nueva dirección */
16
17     return 0; /* indica terminación exitosa */
18
19  } /* fin de main */
```

```
Compiling...
fig07_13.c
C:\Documents and Settings\Laura\Configuración local\Temp\fig07_13.c(15) : error
C2166: l-value specifies const object
Error executing cl.exe.

fig07_13.exe - 1 error(s), 0 warning(s)
```

**Figura 7.13** Intento de modificar un apuntador constante a un dato no constante. (Parte 2 de 2.)

apuntador **`ptr`** se define en la línea 12 como de tipo **`int *const`**. La definición se lee de derecha a izquierda como "**`ptr`** es un apuntador constante a un entero". El apuntador se inicializa (línea 12) con la dirección de la variable entera **`x`**. El programa intenta asignar la dirección de **`y`** a **`ptr`** (línea 15), pero se genera un mensaje de error.

***Intento de modificar un apuntador constante a un dato constante***

El menor privilegio de acceso lo tiene un apuntador constante a un dato constante. Tal apuntador apunta a la misma dirección de memoria, y no se puede modificar el dato en dicha ubicación de memoria. Ésta es la manera como se debe pasar un arreglo a una función que sólo ve al arreglo mediante la notación de subíndices de arreglos y que no lo modifica. La figura 7.14 define una variable apuntador **`ptr`** (línea 13) de tipo **`const int *const`**, lo cual se lee de derecha a izquierda como "**`ptr`** es un apuntador constante a un entero constante". La figura muestra los mensajes de error que se generan cuando intentamos modificar el dato al cual apunta **`ptr`** (línea 17), y cuando intentamos modificar la dirección almacenada en la variable apuntador (línea 18).

```
1   /* Figura 7.14: fig07_14.c
2      Intenta modificar un apuntador constante a un dato constante. */
3   #include <stdio.h>
4
5   int main()
6   {
7      int x = 5; /* inicializa x */
8      int y;     /* define y */
9
10   /* ptr es un apuntador constante a un entero constante. ptr siempre
11        apunta a la misma ubicación; el entero en esa ubicación
12        no se puede modificar */
13     const int *const ptr = &x;
14
```

**Figura 7.14** Intento de modificar un apuntador constante a un dato constante. (Parte 1 de 2.)

```
15  printf( "%d\n", *ptr );
16
17     *ptr = 7; /* error: *ptr es const; no se puede asignar un nuevo valor */
18     ptr = &y; /* error: ptr es const; no se puede asignar una nueva dirección */
19
20     return 0; /* indica terminación exitosa */
21
22  } /* fin de main */
```

```
Compiling...
fig07_14.c
C:\Documents and Settings\Laura\Configuración local\Temp\fig07_14.c(17) : error
C2166: l-value specifies const object
C:\Documents and Settings\Laura\Configuración local\Temp\fig07_14.c(18) : error
C2166: l-value specifies const object
Error executing cl.exe.

fig07_14.exe - 2 error(s), 0 warning(s)
```

**Figura 7.14** Intento de modificar un apuntador constante a un dato constante. (Parte 2 de 2.)

## 7.6 Ordenamiento de burbuja mediante llamadas por referencia

Modifiquemos el programa de ordenamiento de burbuja de la figura 6.15 para utilizar dos funciones, **ordenaMBurbuja** e **intercambia**. La función **ordenaMBurbuja** ordena el arreglo. Ésta invoca a la función **intercambia** (línea 53) para intercambiar los elementos del arreglo **arreglo[j]** y del **arreglo [j + 1]** (vea la figura 7.15). Recuerde que C promueve el ocultamiento de información entre las funciones, de manera que **intercambia** no tiene acceso a los elementos individuales del arreglo en **ordenaMBurbuja**. Debido a que **ordenaMBurbuja** *quiere* **intercambiar** para tener acceso a los elementos del arreglo que se van a intercambiar, **ordenaMBurbuja** pasa cada uno de estos elementos a **intercambia** mediante una llamada por referencia; la dirección de cada elemento del arreglo se pasa de manera explícita. Aunque los arreglos completos se pasan automáticamente por referencia, los elementos individuales del arreglo son escalares, y normalmente se pasan por valor. Por lo tanto, **ordenaMBurbuja** utiliza el operador de dirección (**&**) en cada uno de los elementos del arreglo en la llamada a **intercambia** (línea 53) de la siguiente manera

```
intercambia( &arreglo[ j ], &arreglo[ j + 1 ] );
```

para efectuar la llamada por referencia. La función **intercambia** recibe **&arreglo[ j ]** en la variable apuntador **ptrElemento1** (línea 64). Incluso cuando **intercambia** (debido al ocultamiento de información) no está autorizada para conocer el nombre de **arreglo[ j ]**, ésta puede utilizar ***ptrElemento1** como un sinónimo para **arreglo[ j ]**. Por lo tanto, cuando **intercambia** hace referencia a ***ptrElemento1**, en realidad hace referencia a **arreglo[ j ]** en **ordenaMBurbuja**. De manera similar, cuando **intercambia** hace referencia a ***ptrElemento2**, en realidad hace referencia a **arreglo[ j + 1 ]** en **ordenaMBurbuja**. Incluso cuando **intercambia** no está autorizado para decir

```
mantiene = arreglo[ j ];
arreglo[ j ] = arreglo[ j + 1 ];
arreglo[ j + 1 ] = mantiene;
```

se obtiene precisamente el mismo efecto en las líneas 66 a 68

```
int mantiene = *ptrElemento1;
*ptrElemento1 = *ptrElemento2;
*ptrElemento2 = mantiene;
```

en la función **intercambia** de la figura 7.15

```
/* Figura 7.15: fig07_15.c
   Este programa coloca valores dentro de un arreglo, ordena los valores en
   orden ascendente, e imprime los resultados del arreglo. */
#include <stdio.h>
#define TAMANIO 10

void ordenaMBurbuja( int * const arreglo, const int tamanio ); /* prototipo */

int main()
{
   /* inicializa el arreglo a */
   int a[ TAMANIO ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   int i; /* contador */

   printf( “Elementos de datos en el orden original\n” );

   /* ciclo a través del arreglo a */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%4d”, a[ i ] );
   } /* fin de for */

   ordenaMBurbuja( a, TAMANIO ); /* ordena el arreglo */

   printf( “\nElementos de datos en orden ascendente\n” );

   /* ciclo a través del arreglo a */
   for ( i = 0; i < TAMANIO; i++ ) {
      printf( “%4d”, a[ i ] );
   } /* fin de for */

   printf( “\n” );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* ordena un arreglo de enteros mediante el algoritmo de la burbuja */
void ordenaMBurbuja( int * const arreglo, const int tamanio )
{
   void intercambia( int *ptrElemento1, int *ptrElemento2 ); /* prototipo */
   int pasada; /* contador de pasadas */
   int j;      /* contador de comparaciones */

   /* ciclo para controlar las pasadas */
   for ( pasada = 0; pasada < tamanio - 1; pasada++ ) {

      /* ciclo para controlar las comparaciones durante cada pasada */
      for ( j = 0; j < tamanio - 1; j++ ) {

         /* intercambia los elementos adyacentes, si no están en orden */
         if ( arreglo[ j ] > arreglo[ j + 1 ] ) {
            intercambia( &arreglo[ j ], &arreglo[ j + 1 ] );
         } /* fin de if */

```

**Figura 7.15** Ordenamiento de burbuja mediante una llamada por referencia. (Parte 1 de 2.)

```
56         } /* fin del for interno */
57
58      } /* fin del for externo */
59
60   } /* fin de la función ordenaMBurbuja */
61
62   /* intercambia los valores en las ubicaciones de memoria a los cuales
        apunta ptrElemento1 y
63      ptrElemento2 */
64   void intercambia( int *ptrElemento1, int *ptrElemento2 )
65   {
66      int almacena = *ptrElemento1;
67      *ptrElemento1 = *ptrElemento2;
68      *ptrElemento2 = almacena;
69   } /* fin de la función intercambia */
```

```
Elementos de datos en el orden original
   2   6   4   8  10  12  89  68  45  37
Elementos de datos en orden ascendente
   2   4   6   8  10  12  37  45  68  89
```

**Figura 7.15** Ordenamiento de burbuja mediante una llamada por referencia. (Parte 2 de 2.)

Debemos observar varias características de la función **`ordenaMBurbuja`**. El encabezado de la función (línea 39) declara arreglo como **`int *arreglo`** en lugar de **`int arreglo[]`**, para indicar que **`ordenaMBurbuja`** recibe un arreglo con un solo subíndice como argumento (de nuevo, estas notaciones son intercambiables). El parámetro **`tamanio`** se declara como **`const`** para promover el principio del menor privilegio. Aunque el parámetro **`tamanio`** recibe una copia del valor en **`main`**, y al modificar la copia no puede cambiar el valor en **`main`**, **`ordenaMBurbuja`** no necesita alterar **`tamanio`** para llevar a cabo su tarea. El tamaño del arreglo permanece fijo durante la ejecución de la función **`ordenaMBurbuja`**. Por lo tanto, **`tamanio`** se declara como **`const`** para garantizar que no se modifique. Si el tamaño del arreglo se modifica durante el proceso de ordenamiento, al algoritmo de ordenamiento podría no ejecutarse correctamente.

El prototipo para la función **`intercambia`** (línea 41) se incluye en el cuerpo de la función **`ordenaMBurbuja`**, debido a que ésta es la única función que llama a **`intercambia`**. Colocar el prototipo dentro de **`ordenaMBurbuja`** restringe las propias llamadas de **`intercambia`** a aquellas que se hagan desde **`ordenaMBurbuja`**. Otras funciones que intenten llamar a **`intercambia`** no tienen acceso al prototipo adecuado, de modo que el compilador genera uno automáticamente. Por lo general, esto produce un prototipo que no coincide con el encabezado de la función (y genera un error de compilación) debido a que el compilador asume un tipo de retorno **`int`** para el tipo de los parámetros.



### Observación de ingeniería de software 7.3

*Colocar los prototipos de las funciones en la definición de otras funciones promueve el principio del menor privilegio, al restringir las llamadas a las funciones, a aquellas en donde aparece su prototipo.*

Observe que la función **`ordenaMBurbuja`** recibe el tamaño del arreglo como un parámetro (línea 39). La función debe saber el tamaño del arreglo para ordenarlo. Cuando se pasa un arreglo a la función, ésta recibe la dirección de memoria del primer elemento del arreglo. Por supuesto, la dirección no coincide con el número de elementos del arreglo. Por lo tanto, el programador debe pasar el tamaño del arreglo a la función.

En el programa, el tamaño del arreglo se pasa de manera explícita a la función **`ordenaMBurbuja`**. Existen dos beneficios principales en este método, la reutilización de software y la ingeniería de software apropiada. Al definir a la función para que reciba el tamaño del arreglo como argumento, permitimos a cualquier programa que utilice la función para ordenar arreglos enteros con un solo subíndice de cualquier tamaño.



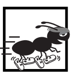
### Observación de ingeniería de software 7.4

*Cuando pase un arreglo a una función, también pase el tamaño del arreglo. Esto ayuda a hacer a la función reutilizable en muchos programas.*

También podríamos haber almacenado el tamaño del arreglo dentro de una variable global que fuera accesible para todo el programa. Esto sería más eficiente debido a que no se hace una copia del tamaño para pasarla a la función. Sin embargo, otros programas que requieren la capacidad de ordenamiento de arreglos enteros podrían no contar con la variable global, de manera que la función no podría utilizarse en dichos programas.

**Observación de ingeniería de software 7.5**

*A menudo, las variables globales violan el principio del menor privilegio y pueden provocar una pobre ingeniería de software.*



**Tip de rendimiento 7.2**

*Pasar el tamaño de un arreglo a una función toma tiempo y requiere espacio adicional en la pila, debido a que se crea una copia del tamaño para pasarla a la función. Las variables globales no requieren tiempo o espacio adicional, debido a que cualquier función puede acceder a ellas de manera directa.*

El tamaño del arreglo pudo programarse de manera directa dentro de la función. Esto restringe el uso de la función a un arreglo de un tamaño específico, y reduce de manera significativa su reutilización. Sólo los programas que procesan arreglos enteros con un solo subíndice y del tamaño específico podrán utilizar esta función.

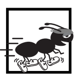
## 7.7 El operador `sizeof`

C proporciona el operador unario **`sizeof`** para determinar el tamaño en bytes de un arreglo (o de cualquier otro tipo de dato) durante la compilación del programa. Cuando se aplica al nombre de un arreglo como en la figura 7.16 (línea 14), el operador **`sizeof`** devuelve el número total de bytes del arreglo como un entero. Observe que, por lo general, las variables de tipo **`float`** se almacenan en 4 bytes de memoria, y que un **`arreglo`** se define para contener 20 elementos. Por lo tanto, existe un total de 80 bytes en el **`arreglo`**.

```
1  /* Figura 7.16: fig07_16.c
2     Cuando el operador sizeof se utiliza en un nombre de arreglo,
3     éste devuelve el número de bytes en el arreglo. */
4  #include <stdio.h>
5
6  size_t obtieneTamanio( float *ptr ); /* prototipo */
7
8  int main()
9  {
10    float arreglo[ 20 ]; /* crea arreglo */
11
12    printf( "El número de bytes en el arreglo es %d"
13            "\nEl número de bytes devueltos por obtieneTamanio es %d\n",
14            sizeof( arreglo ), obtieneTamanio( arreglo ) );
15
16    return 0; /* indica terminación exitosa */
17
18 } /* fin de main */
19
20 /* devuelve el tamaño de ptr */
21 size_t obtieneTamanio( float *ptr )
22 {
23    return sizeof( ptr );
24
25 } /* fin de la función obtieneTamanio */
```

```
El número de bytes en el arreglo es 80
El número de bytes devueltos por obtieneTamanio es 4
```

**Figura 7.16** Cuando el operador **`sizeof`** se aplica al nombre de un arreglo, éste devuelve el número de bytes del arreglo.



**Tip de rendimiento 7.3**

***sizeof*** *es un operador en tiempo de compilación, de manera que no implica sobrecarga alguna en tiempo de ejecución.*

También se puede determinar el número de elementos del arreglo mediante **sizeof**. Por ejemplo, considere la siguiente definición de un arreglo:

```
double real[ 22 ];
```

Por lo general, las variables de tipo **double** se almacenan en 8 bytes de memoria. Entonces, el arreglo **real** contiene un total de 176 bytes. Para determinar el número de elementos en el arreglo, podemos utilizar la siguiente expresión:

```
sizeof( real ) / sizeof( double )
```

La expresión determina el número de bytes del arreglo **real** y lo divide entre el número de bytes utilizados en memoria para almacenar un valor **double**.

Observe que el tipo de retorno de la función **obtieneTamanio** es **size_t**. El tipo **size_t** es un tipo definido por el C estándar como el tipo entero (con signo o sin signo) del valor que devuelve el operador **sizeof**. El tipo **size_t** se define en el encabezado **<stddef.h>** (el cual se incluye en varios encabezados, tales como **<stdio.h>**). La figura 7.17 calcula el número de bytes que se utilizan para almacenar cada uno de los tipos de datos estándares. Los resultados pueden variar entre computadoras.

```
/* Figura 7.17: fig07_17.c
   Demostración del operador sizeof */
#include <stdio.h>

int main()
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int arreglo[ 20 ];  /* crea el arreglo de 20 elementos int */
   int *ptr = arreglo; /* crea el apuntador al arreglo */

   printf( “      sizeof c = %d\tsizeof(char)  = %d”
           “\n    sizeof s = %d\tsizeof(short) = %d”
           “\n    sizeof i = %d\tsizeof(int) = %d”
           “\n    sizeof l = %d\tsizeof(long) = %d”
           “\n    sizeof f = %d\tsizeof(float) = %d”
           “\n    sizeof d = %d\tsizeof(double) = %d”
           “\n    sizeof ld = %d\tsizeof(long double) = %d”
           “\n    sizeof arreglo = %d”
           “\n    sizeof ptr = %d\n”,
           sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
           sizeof( int ), sizeof l, sizeof( long ), sizeof f,
           sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
           sizeof( long double ), sizeof arreglo, sizeof ptr );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 7.17** Uso del operador **sizeof** para determinar los tamaños de los tipos de datos estándares. (Parte 1 de 2.)

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
sizeof arreglo = 80
   sizeof ptr = 4
```

**Figura 7.17** Uso del operador **sizeof** para determinar los tamaños de los tipos de datos estándares. (Parte 2 de 2.)



**Tip de portabilidad 7.2**

*El número de bytes que se utilizan para almacenar un tipo de dato en particular puede variar entre sistemas. Cuando escriba programas que dependan del tamaño del tipo de dato y que se ejecutarán en varios sistemas de computadoras, utilice* ***sizeof*** *para determinar el número de bytes requeridos para almacenar los tipos de datos.*

El operador **sizeof** se puede aplicar a cualquier nombre de variable, tipo o valor (incluso el valor de una expresión). Cuando se aplica al nombre de una variable (que no es el nombre de un arreglo) o a una constante, devuelve el número de bytes que se utilizan para almacenar un tipo de variable o constante específica. Observe que los paréntesis utilizados con **sizeof** son requeridos si se proporciona el tipo de dato como operando. En este caso, omitir el paréntesis provoca un error de sintaxis. No se requieren los paréntesis si se proporciona un nombre de variable como operando.

## 7.8 Expresiones con apuntadores y aritmética de apuntadores

Los apuntadores son operandos válidos dentro de las expresiones aritméticas, expresiones de asignación y expresiones de comparación. Sin embargo, por lo general no todos los operadores utilizados son válidos con el uso de las variables de apuntadores. Esta sección describe los operadores que pueden tener apuntadores como operandos, y cómo se utilizan estos operadores.

Se puede realizar un conjunto limitado de operaciones con los apuntadores. Un apuntador se puede *incrementar*(**++**) o *decrementar*(**--**), *se puede sumar un entero a un apuntador* (**+** o **+=**), *se puede restar un entero a un apuntador* (**-** o **-=**) y *se puede restar un apuntador a otro*.

Suponga que el arreglo **int v[ 5 ]** ya está definido y que su primer elemento se encuentra en la ubicación 3000 de memoria. Suponga que el apuntador **ptrV** se inicializa para apuntar a **v[ 0 ]**, es decir, el valor de **ptrV** es **3000**. La figura 7.18 ilustra esta situación para una máquina con enteros de 4 bytes. Observe que **ptrV** puede inicializarse para que apunte al arreglo v con cualquiera de las instrucciones

```
ptrV = v;
ptrV = &v[ 0 ];
```

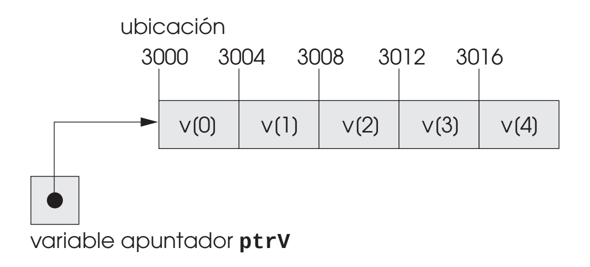



**Figura 7.18** El arreglo **v** y la variable **ptrV** que apuntan a **v**.



**Tip de portabilidad 7.3**

*La mayoría de las computadoras actuales tienen enteros de 2 y 4 bytes. Algunas de las máquinas más nuevas utilizan enteros de 8 bytes. Debido a que los resultados de la aritmética de apuntadores dependen del tamaño de los objetos al que apunta el apuntador, la aritmética de apuntadores depende de la máquina.*

En la aritmética convencional, **3000 + 2** da como resultado **3002**. Por lo general, éste no es el caso en la *aritmética de apuntadores*. Cuando se suma o se resta un entero o a un apuntador, el apuntador no aumenta o disminuye por dicho entero, sino por el número de veces del tamaño del objeto al que hace referencia el apuntador. El número de bytes depende del tipo de datos del objeto. Por ejemplo, la instrucción

```
ptrV += 2;
```

producirá **3008 (3000 + 2 * 4)**, suponiendo que un entero se almacena en 4 bytes de memoria. En el arreglo **v**, **ptrV** ahora apunta a **v[ 2 ]** (figura 7.19). Si un entero se almacena en 2 bytes de memoria, entonces el cálculo anterior arrojará la dirección de memoria **3004 (3000 + 2*2**). Si el arreglo es de un tipo de dato diferente, la instrucción anterior incrementará el apuntador el doble del número de bytes necesarios para almacenar un objeto de ese tipo de dato. Cuando utilizamos la aritmética de apuntadores en un arreglo de caracteres, los resultados serán consistentes con la aritmética regular, debido a que cada carácter ocupa 1 byte de longitud.

Si **ptrV** se incrementa a **3016**, lo cual apunta a **v[ 4 ],** la instrucción

```
ptrV -= 4;
```

establece **ptrV** de nuevo en **3000**, es decir, al principio del arreglo. Si un apuntador se incrementa o se decrementa en uno, pueden utilizarse los operadores de incremento (**++**) y decremento(**--**). Cualquiera de las instrucciones

```
++ptrV
ptrV++
```

incrementan el apuntador para que apunte al elemento previo del arreglo; o cualquiera de las instrucciones

```
--ptrV;
ptrV--;
```

decrementan el apuntador para que apunte al elemento previo del arreglo.

Las variables apuntador se pueden restar entre sí. Por ejemplo, si **ptrV** contiene la ubicación **3000**, y **ptrV2** contiene la dirección **3008**, la instrucción

```
x = ptrV2 – ptrV;
```

asignará a **x** el número de elementos del arreglo **ptrV** a **ptrV2**, en este caso **2** (y no 8). La aritmética de apuntadores no tiene sentido a menos que se realice en un arreglo. No podemos asumir que dos variables del mismo tipo se almacenan de manera contigua en memoria, a menos que sean elementos adyacentes de un arreglo.



**Error común de programación 7.5**

*Utilizar la aritmética de apuntadores sobre un apuntador que no hace referencia a un elemento de un arreglo.*

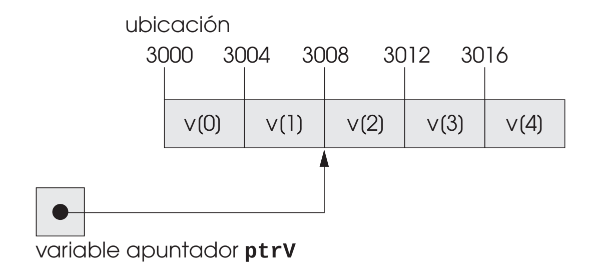



**Figura 7.19** El apuntador **ptrV** después de aplicar la aritmética de apuntadores.

**Error común de programación 7.6**



*Restar o comparar dos apuntadores que no hacen referencia a los elementos del mismo arreglo.*

**Error común de programación 7.7**

*Rebasar el final de un arreglo cuando se utiliza la aritmética de apuntadores.*

Un apuntador puede asignarse a otro apuntador si ambos son del mismo tipo. La excepción a esta regla es un *apuntador a* ***void*** (es decir, ***void ****), el cual es un apuntador genérico que puede representar a cualquier tipo de apuntador. Todos los tipos de apuntadores pueden asignarse al apuntador **void**, y el apuntador **void** puede asignarse a todos los tipos de apuntadores. En ambos casos, no es necesario un operador de conversión de tipo.

No se puede desreferenciar un apuntador a **void**. Por ejemplo, el compilador sabe que un apuntador a **int** hace referencia a cuatro bytes de memoria en una máquina con enteros de 4 bytes, pero un apuntador a **void** simplemente contiene una ubicación de memoria para un tipo de dato desconocido, el compilador no puede saber con precisión el número de bytes al cual hace referencia. El compilador debe saber el tipo de dato para determinar el número de bytes que se van a desreferenciar en un apuntador en especial.

**Error común de programación 7.8**



*Asignar un apuntador de un tipo específico a un apuntador de otro tipo, incluso si es de tipo* ***void ****, *es un error de sintaxis.*

**Error común de programación 7.9**



*Desreferenciar un apuntador* ***void ****, *es un error de sintaxis.*

Los apuntadores se pueden comparar por medio de los operadores de igualdad y de relación, pero tales comparaciones son irrelevantes, a menos que los apuntadores apunten a los elementos del mismo arreglo. Las comparaciones entre apuntadores comparan las direcciones almacenadas en los apuntadores. Por ejemplo, una comparación entre dos apuntadores que apuntan a elementos del mismo arreglo puede mostrar que uno de ellos apunta al elemento con el número más alto del arreglo. Un uso común de la comparación entre apuntadores es determinar si un apuntador es **NULL**.

## 7.9 Relación entre apuntadores y arreglos

En C, los arreglos y los apuntadores están íntimamente relacionados, y a menudo se pueden utilizar de manera indistinta. Un nombre de arreglo puede interpretarse como un apuntador constante. Los apuntadores se pueden utilizar para realizar cualquier operación que involucre subíndices de arreglos.

Suponga que el arreglo de enteros **b[ 5 ]** y la variable apuntador **ptrB** ya están definidos. Dado que el nombre del arreglo (sin subíndice) es un apuntador al primer elemento del mismo arreglo, podemos establecer **ptrB** igual a la dirección del primer elemento del arreglo **b** mediante la instrucción

```
ptrB = b;
```

Esta instrucción es equivalente a tomar la dirección del primer elemento del arreglo de la siguiente manera

```
ptrB = &b[ 0 ];
```

De manera alterna, se puede hacer referencia al elemento **b[ 3 ]** del arreglo mediante la expresión con apuntadores

```
*( ptrB + 3 )
```

El **3** en la expresión de arriba es el *desplazamiento* del apuntador. Cuando el apuntador apunta hacia el principio de un arreglo, el desplazamiento indica a cuál elemento del arreglo se debe hacer referencia, y el valor de desplazamiento es idéntico al subíndice del arreglo. A la notación anterior se le conoce como *notación apuntador/desplazamiento*. Los paréntesis son necesarios debido a que la precedencia de ***** es más alta que la precedencia de **+**. Sin los paréntesis, la expresión de arriba sumaría **3** al valor de la expresión ***ptrB** (es decir, se sumarian **3** a **b[ 0 ]**, suponiendo que **ptrB** apunta al principio del arreglo). Tal como se puede hacer referencia al elemento del arreglo mediante una expresión de apuntador, la dirección

```
&b[ 3 ]
```

puede escribirse mediante la expresión de apuntador

```
ptrB + 3
```

El arreglo mismo puede tratarse como un apuntador y utilizarse en la aritmética de apuntadores. Por ejemplo, la expresión

```
*( b + 3 )
```

también hace referencia al elemento **b[ 3 ]** del arreglo. Por lo general, todas las expresiones de arreglos con subíndices pueden escribirse mediante un apuntador y un desplazamiento. En este caso, la notación apuntador/desplazamiento se utilizó con el nombre del arreglo como un apuntador. Observe que la instrucción anterior no modifica el nombre del arreglo de manera alguna; **b** aún apunta al primer elemento del arreglo.

A los apuntadores se les puede asignar subíndices tal como a los arreglos. Por ejemplo, si **ptrB** tiene el valor **b**, la expresión

```
ptrB[ 1 ]
```

hace referencia al elemento **b[ 1 ]**. A esto se le llama *notación apuntador/subíndice*.

Recuerde que el nombre del arreglo es esencialmente un apuntador constante; siempre apunta al principio del arreglo. Entonces, la expresión

```
b += 3
```

es inválida debido a que intenta modificar el valor del nombre del arreglo mediante la aritmética de apuntadores.



**Error común de programación 7.10**

*Intentar modificar el nombre del arreglo con aritmética de apuntadores, es un error de sintaxis.*

La figura 7.20 utiliza los cuatro métodos que explicamos para hacer referencia a los elementos de un arreglo: subíndices de arreglos, apuntador/desplazamiento con el nombre del arreglo como apuntador, *subíndices de apuntadores*, y apuntador/desplazamiento con un apuntador, para imprimir los cuatro elementos del arreglo entero **b**.

```
/* Figura 7.20: fig07_20.cpp
   Uso de las notaciones de subíndices y de apuntadores con arreglos */

#include <stdio.h>

int main()
{
   int b[] = { 10, 20, 30, 40 }; /* inicializa el arreglo b */
   int *ptrB = b;                /* establece ptrB para que apunte
   al arreglo b */
   int i;                        /* contador */
   int desplazamiento;           /* contador */

   /* muestra el arreglo b con la notación de subíndices */
   printf( "Arreglo b impreso con:\nNotacion de subindices de arreglos\n" );

   /* ciclo a través del arreglo b*/
   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   } /* fin de for */

   /* muestra el arreglo b mediante el uso del nombre del arreglo y
      notación apuntador/desplazamiento */
```

**Figura 7.20** Uso de los cuatro métodos para hacer referencia a los elementos de un arreglo. (Parte 1 de 2.)

```
    printf( "\nNotacion apuntador/desplazamiento donde\n"
            "el apuntador es el nombre del arreglo\n" );

    /* ciclo a través del arreglo b */
    for ( desplazamiento = 0; desplazamiento < 4; desplazamiento++ ) {
       printf( "*( b + %d ) = %d\n", desplazamiento, *( b + desplazamiento ) );
    } /* fin de for */

    /* muestra el arreglo b mediante el uso de ptrB y notación de subíndices
       de arreglos */
    printf( "\nNotacion de subindices de arreglos\n" );

    /* ciclo a través del arreglo b */
    for ( i = 0; i < 4; i++ ) {
       printf( "ptrB[ %d ] = %d\n", i, ptrB[ i ] );
    } /* fin de for */

    /* muestra el arreglo b mediante el uso de ptrB y notación de
       apuntador/desplazamiento */
    printf( "\nNotación apuntador desplazamiento\n" );

    /* ciclo a través del arreglo b */
    for ( desplazamiento = 0; desplazamiento < 4; desplazamiento++ ) {
       printf( "*( ptrB + %d ) = %d\n", desplazamiento, *( ptrB + desplazamiento ) );
    } /* fin de for */

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Arreglo b impreso con:
Notacion de subindices de arreglos
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Notacion apuntador/desplazamiento donde
el apuntador es el nombre del arreglo
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Notacion de subindices de arreglos
ptrB[ 0 ] = 10
ptrB[ 1 ] = 20
ptrB[ 2 ] = 30
ptrB[ 3 ] = 40

Notacion apuntador/desplazamiento
*( ptrB + 0 ) = 10
*( ptrB + 1 ) = 20
*( ptrB + 2 ) = 30
*( ptrB + 3 ) = 40
```

**Figura 7.20** Uso de los cuatro métodos para hacer referencia a los elementos de un arreglo. (Parte 2 de 2.)

Para ilustrar con más detalle la posibilidad de intercambiar arreglos y apuntadores, revisemos las dos funciones para copiar cadenas, **copia1** y **copia2**, del programa de la figura 7.21. Ambas funciones copian una cadena (posiblemente un arreglo de caracteres) dentro de un arreglo de caracteres. Después de comparar los prototipos de las funciones para **copia1** y **copia2**, las funciones parecen idénticas. Llevan a cabo la misma tarea; sin embargo, su implementación es diferente.

```
 /* Figura 7.21: fig07_21.c
    Copia de una cadena por medio de la notación de arreglos y la notación
    de apuntadores */
#include <stdio.h>

void copia1( char *s1, const char *s2 ); /* prototipo */
void copia2( char *s1, const char *s2 ); /* prototipo */

int main()
{
   char cadena1[ 10 ];          /* crea el arreglo cadena1 */
   char *cadena2 = “Hola”;      /* crea un apuntador a una cadena */
   char cadena3[ 10 ];          /* crea el arreglo cadena3 */
   char cadena4[] = “Adios”;    /* crea un apuntador a una cadena */

   copia1( cadena1, cadena2 );
   printf( “cadena1 = %s\n”, cadena1 );

   copia2( cadena3, cadena4 );
   printf( “cadena3 = %s\n”, cadena3 );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* copia s2 en s1 con el uso de la notación de arreglos */
void copia1( char *s1, const char *s2 )
{
   int i; /* contador */

   /* realiza el ciclo a través de la cadena */
   for ( i = 0; ( s1[ i ] = s2[ i ] ) != ‘\0’; i++ ) {
      ;   /* no realiza tarea alguna en el cuerpo */
   } /* fin de for */

} /* fin de la función copia1 */

/* copia s2 en s1 con el uso de la notación de apuntadores */
void copia2( char *s1, const char *s2 )
{
   /* realiza el ciclo a través de las cadenas */
   for ( ; ( *s1 = *s2 ) != ‘\0’; s1++, s2++ ) {
      ;   /* no realiza tarea alguna en el cuerpo */
   } /* fin de for */

} /* fin de la función copia2 */
```

```
cadena1 = Hola
cadena3 = Adios
```

**Figura 7.21** Copia de una cadena mediante la notación de arreglos y la notación de apuntadores.

La función **copia1** utiliza la notación de subíndices de arreglos para copiar la cadena de **s2** en la cadena de caracteres **s1**. La función define una variable entera como contador, **i**, como el subíndice del arreglo. El encabezado de la instrucción **for** (línea 31) realiza la operación completa de copia; su cuerpo es la instrucción vacía. El encabezado especifica que **i** se inicializa en cero y se incrementa en **1** en cada iteración del ciclo. La condición de la instrucción **for**, **s1[ i ] = s2[ i ]**, realiza la operación de copiar carácter por carácter desde **s2** a **s1**. Cuando encuentra el carácter nulo en **s2**, se asigna a **s1**, y el valor de la asignación se convierte en el valor asignado al operador de la izquierda **(s1)**. El ciclo termina debido a que el valor entero del carácter nulo es cero (falso).

La función **copia2** utiliza apuntadores y la aritmética de apuntadores para copiar la cadena de **s2** al arreglo de caracteres **s1**. De nuevo, el encabezado de la instrucción **for** (línea 41) realiza la operación completa de copia. El encabezado no incluye variable alguna de inicialización. Al igual que en la función **copia1**, la condición **(*s1 = *s2)** realiza la operación de copia. Se desreferencia el apuntador de copia **s2**, y el carácter resultante se asigna al apuntador desreferenciado **s1**. Después de la asignación en la condición, los apuntadores se incrementan para apuntar al siguiente elemento del arreglo **s1** y al siguiente carácter de la cadena **s2**, respectivamente. Cuando se encuentra el carácter nulo en **s2**, se asigna al apuntador desreferenciado **s1** y el ciclo termina.

Observe que el primer argumento tanto de **copia1** como de **copia2** debe ser un arreglo lo suficientemente grande para almacenar la cadena en el segundo argumento. De lo contrario, puede ocurrir un error cuando se intente escribir en una ubicación de memoria que no es parte del arreglo. Además, observe que el segundo parámetro de cada función se declara como **const char *** (una constante cadena). En ambas funciones, el segundo argumento se copia dentro del primer argumento, los caracteres se leen desde ahí, uno a la vez, pero nunca se modifican. Por lo tanto, el segundo parámetro se declara para que apunte a un valor constate y para promover el principio del menor privilegio, ninguna función requiere la capacidad de modificar el segundo argumento, de manera que no se les proporciona esta capacidad.

## 7.10 Arreglos de apuntadores

Los arreglos pueden contener apuntador. Uno de los usos comunes de los *arreglos de apuntadores* es el de formar un *arreglo de cadenas*, llamado también *arreglo cadena*. Cada en elemento en el arreglo es una cadena, pero en C una cadena es, en esencia, un apuntador a su primer carácter. De modo que cada entrada en el arreglo de cadenas es en realidad un apuntador al primer carácter de la cadena. Considere la definición del arreglo de cadenas **palos**, éste podría ser útil para representar las cartas de una baraja.

```
const char *palos[ 4 ] = { “Corazones”, “Diamantes”, “Treboles”, “Espadas” };
```

La parte de la definición de **palos[ 4 ]** indica un arreglo de 4 elementos. La parte **char *** de la declaración indica que cada elemento del arreglo palos es de tipo “apuntador a **char**”. El calificador **const** indica que las cadenas a las que apunta cada elemento apuntador no podrán ser modificadas. Los cuatro valores a colocarse en el arreglo son **“Corazones”**, **“Diamantes”**, **“Treboles”** y **“Espadas”**. Cada uno de ellos se almacena en memoria como una cadena de terminación nula, la cual es un carácter más largo que el número de caracteres entre comillas. Las cuatro cadenas contienen 10, 10, 9 y 8 caracteres de largo, respectivamente. Aunque parece como si estas cadenas se colocaran en el arreglo **palos**, en realidad solamente se almacenan los apuntadores (figura 7.22). Cada apuntador apunta al primer carácter de su cadena correspondiente. Entonces,

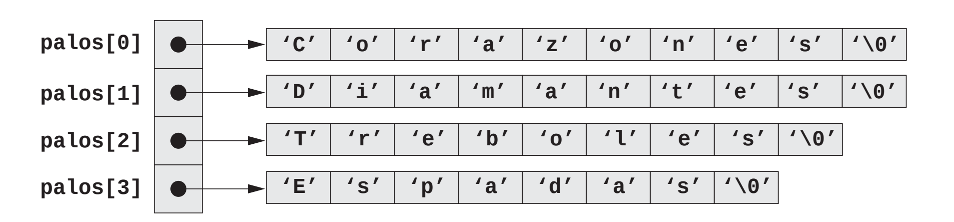



**Figura 7.22** Representación gráfica del arreglo **palos**.

aun cuando el arreglo **palos** tiene un tamaño fijo, proporciona acceso a cadenas de caracteres de cualquier longitud. Esta flexibilidad es un ejemplo de las poderosas capacidades de estructuración de datos en C.

Los palos podrían colocarse en un arreglo de dos dimensiones en el que cada línea representara un palo, y cada columna representara una de las letras del nombre del palo. Tal estructura de datos debiera tener un tamaño fijo de columnas por línea, y ese número tendría que ser tan largo como la cadena más larga. Por lo tanto, podría desperdiciarse una cantidad considerable de memoria si almacenáramos una gran cantidad de cadenas y que la mayoría de éstas fueran menores que la cadena más larga. En la siguiente sección utilizaremos arreglos de cadenas para representar un mazo de cartas.

## 7.11 Ejemplo práctico: Simulación para barajar y repartir cartas

En esta sección, utilizamos la generación de números aleatorios para desarrollar un programa de simulación para barajar y repartir cartas. Este programa puede utilizarse para implementar programas de juegos de cartas específicos. Para poder mostrar algunos pequeños problemas de rendimiento, utilizamos intencionalmente algoritmos para barajar y repartir no tan óptimos. En los ejercicios y en el capítulo 10, desarrollaremos algoritmos más eficientes.

Mediante el método de *mejoramiento arriba-abajo, paso a paso*, desarrollamos un programa que baraja un mazo con 52 cartas de juego, y después reparte cada una de las 52 cartas. El método arriba-abajo es particularmente útil para atacar problemas más complejos que los que hemos visto en los capítulos anteriores.

Utilizaremos un arreglo con dos subíndices de 4 $\times$ 13 elementos para representar el mazo de cartas (figura 7.23). Las filas corresponden a los palos, la fila 0 corresponde a los corazones, la fila 1 corresponde a los diamantes, la fila 2 corresponde a los tréboles y la fila 3 corresponde a las espadas. Las columnas corresponden a las caras de las cartas, las columnas de 0 a 9 corresponden al As y a los números hasta el 10 respectivamente, y las columnas 10 a 12 corresponden al Joto, la Qüina y el Rey, respectivamente. Debemos cargar el arreglo palos con las cadenas que representan los cuatro **palos**, y el arreglo de cadenas con las cadenas de caracteres que representan los trece valores de las **caras**.

El mazo de cartas simulado se puede repartir de la siguiente manera. Primero se inicializa en ceros el arreglo **mazo**. Después, se eligen al azar una **línea** (0-3) y una **columna** (0-12). Se inserta un número 1 al elemento del arreglo **mazo[ línea ][ columna ]** para indicar que esta carta será la primera a repartirse. Este proceso aleatorio continúa con la inserción en el arreglo **mazo** de los números 2, 3,…, 52 para indicar cuáles cartas van a colocarse en segundo, tercero,…, y 52avo lugar del **mazo** barajado. Al comenzar a llenarse el arreglo **mazo** con los números, es posible que una carta se seleccione dos veces, es decir, **mazo[ linea ][ columna ]** será diferente de cero al seleccionarse. Esta selección simplemente se ignora y se eligen aleatoriamente y de manera repetida otras **líneas** y **columnas** hasta que se encuentra una carta no seleccionada. En algún momento, los números del 1 al 52 ocuparán las 52 posiciones del arreglo **mazo**. En este punto, el **mazo** de cartas ya está completamente barajado.

Este algoritmo podría ejecutarse infinitamente si las cartas ya elegidas se eligieran de nuevo de manera aleatoria. A este fenómeno se le conoce como *aplazamiento indefinido*. En los ejercicios, explicaremos un mejor algoritmo para barajar, que elimina la posibilidad del aplazamiento indefinido.

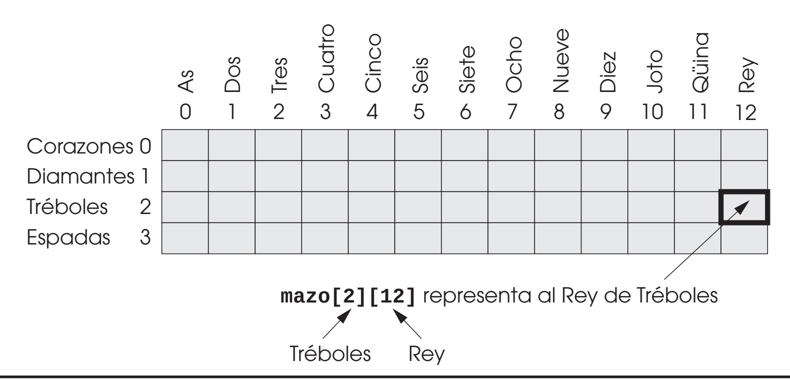
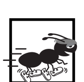



**Figura 7.23** Arreglo con dos subíndices que representa un mazo de cartas.



### Tip de rendimiento 7.4

*Algunas veces un algoritmo que emerge de manera "natural" puede contener sutiles problemas de rendimiento, tales como el aplazamiento indefinido. Busque algoritmos que eviten el aplazamiento indefinido.*

Para repartir la primera carta, buscamos elemento que sea igual a **`1`** en el arreglo **`mazo[ línea ][ columna ]`**. Esto se lleva a cabo anidando instrucciones **`for`** que varíen las líneas de 0 a 3 y las **`columnas`** de 0 a 12. ¿A qué elemento del arreglo corresponde? El arreglo **`palos`** ya se cargó con los cuatro palos, así que para obtener el palo, imprimimos la cadena de caracteres **`palos[ columna ]`**. De manera similar, para obtener el valor de la cara de la carta, imprimimos la cadena de caracteres **`cara[ columna ]`**. También imprimimos la cadena de caracteres **`"de"`**. La impresión de esta información en el orden apropiado nos permite imprimir cada carta en la forma **`"Rey de Treboles"`**, **`"As de Diamantes"`** y así sucesivamente.

Procedamos con el método arriba-abajo y el refinamientopaso a paso. La cima es simplemente

*Baraja y reparte 52 cartas*

Nuestro primer refinamiento arroja:

*Inicializa el arreglo palos*
*Inicializa el arreglo caras*
*Inicializa el arreglo mazo*
*Baraja el mazo*
*Reparte las 52 cartas*

"Baraja el mazo" puede expandirse de la siguiente manera:

*Para cada una de las 52 cartas*
    *Coloca el número de la carta en una posición aleatoria y desocupada del mazo*

"Reparte las 52 cartas" puede expandirse de la siguiente manera:

*Para cada una de las 52 cartas*
    *Encuentra el número de la carta e imprime la cara y el palo de ésta*

Al incorporar estas expansiones tenemos nuestro segundo refinamiento:

*Inicializa el arreglo palos*
*Inicializa el arreglo caras*
*Inicializa el arreglo mazo*

*Para cada una de las 52 cartas*
    *Coloca el número de la carta en una posición aleatoria y desocupada del mazo*

*Para cada una de las 52 cartas*
    *Encuentra el número de la carta e imprime la cara y el palo de ésta*

"Coloca el número de la carta en una posición aleatoria y desocupada del mazo" puede expandirse de la siguiente manera:

*Elige aleatoriamente la posición del mazo*

*Mientras la posición elegida haya sido previamente seleccionada*
    *Elige aleatoriamente la posición del mazo*

*Coloca el número de la carta en la posición del mazo*

"Encuentra el número de la carta e imprime la cara y el palo de ésta" puede expandirse de la siguiente manera:

*Para cada posición del arreglo mazo*
    *Si la posición contiene el número de carta deseado*
        *Imprime la cara y el mazo de la carta*

Al incorporar estas expansiones tenemos nuestro tercer Refinamiento:

*Inicializa el arreglo palos*
*Inicializa el arreglo caras*
*Inicializa el arreglo mazo*

*Para cada una de las 52 cartas*
    *Elige aleatoriamente la posición del mazo*

    *Mientras posición elegida haya sido previamente seleccionada*
        *Elige aleatoriamente la posición del mazo*

    *Coloca el número de la carta en la posición del mazo*

*Para cada una de las 52 cartas*
    *Para cada posición del arreglo mazo*
        *Si la posición contiene el número de carta deseado*
            *Imprime la cara y el mazo de la carta*

Esto completa el proceso de refinamiento. Observe que este programa es más eficiente si las porciones barajar y repartir del algoritmo se combinan para que cada carta tal como está colocada en el mazo. Elegimos programar estas operaciones por separado, debido a que por lo general las cartas se reparten después de que ya se barajaron (y no mientras se barajan).

En la figura 7.24 mostramos el programa para barajar y repartir, y en la 7.25 un ejemplo de su ejecución. Observe el uso del especificador de conversión **`%s`** para imprimir cadenas de caracteres en las llamadas a **`printf`**. El argumento correspondiente en la llamada a **`printf`** debe ser un apuntador a **`char`** (o un arreglo **`char`**). En la función reparte, la especificación de formato **`"%6s de %-9s"`** (línea 76) imprime una cadena de caracteres justificada a la derecha en un campo de cinco caracteres, seguido por **`" de "`** y una cadena de caracteres justificada a la izquierda en un campo de nueve caracteres. El signo menos en **`%-9s`** significa que la cadena se justifica a la izquierda en un campo de longitud igual a 9.

```
1  /* Figura 7.24: fig07_24.c
2     Programa para barajar y repartir cartas */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* prototipos */
8  void baraja( int wMazo[][ 13 ] );
9  void reparte( const int wMazo[][ 13 ], const char *wCara[],
10               const char *wPalo[] );
11
12 int main()
13 {
14    /* inicializa el arreglo palo */
15    const char *palo[ 4 ] = { "Corazones", "Diamantes", "Treboles", "Espadas" };
16
17    /* inicializa el arreglo cara */
18    const char *cara[ 13 ] =
19       { "As", "Dos", "Tres", "Cuatro",
20         "Cinco", "Seis", "Siete", "Ocho",
21         "Nueve", "Diez", "Joto", "Quina", "Rey" };
22
23    /* inicializa el arreglo mazo */
24    int mazo[ 4 ][ 13 ] = { 0 };
```

**Figura 7.24** Programa para repartir las cartas. (Parte 1 de 3.)

```

   srand( time( 0 ) ); /* semilla del generador de números aleatorios */

   baraja( mazo );
   reparte( mazo, cara, palo );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* baraja las cartas del mazo */
void baraja( int wMazo[][ 13 ] )
{
   int fila;    /* número de fila */
   int columna; /* número de columna */
   int carta;   /* contador */

   /* elige aleatoriamente un espacio para cada una de las 52 cartas */
   for ( carta = 1; carta <= 52; carta++ ) {

      /* elije una nueva ubicación al azar hasta que encuentra un espacio vacío */
      do {
         fila = rand() % 4;
         columna = rand() % 13;
      } while( wMazo[ fila ][ columna ] != 0 ); /* fin de do...while */

      /* coloca el número de carta en el espacio vacío del mazo */
      wMazo[ fila ][ columna ] = carta;
   } /* fin de for */

} /* fin de la función baraja */

/* reparte las cartas del mazo */
void reparte( const int wMazo[][ 13 ], const char *wCara[],
            const char *wPalo[] )
{
   int carta;   /* contador de cartas */
   int fila;    /* contador de filas */
   int columna; /* contador de columnas */

   /* reparte cada una de las 52 cartas */
 for ( carta = 1; carta <= 52; carta++ ) {

      /* realiza el ciclo a través de las filas de wMazo */
      for ( fila = 0; fila <= 3; fila++ ) {

         /* realiza el ciclo a través de las columnas de wMazo en la fila actual */
         for ( columna = 0; columna <= 12; columna++ ) {

            /* si el espacio contiene la carta actual, despliega la carta */
            if ( wMazo[ fila ][ columna ] == carta ) {
               printf( “%6s de %-9s%c”, wCara[ columna ], wPalo[ fila ],
                  carta % 2 == 0 ? ‘\n’ : ‘\t’ );
            } /* fin de if */

```

**Figura 7.24** Programa para repartir las cartas. (Parte 2 de 3.)

```
            } /* fin de for */

         } /* fin de for */

      } /* fin de for */

   } /* fin de la función reparte */
```

**Figura 7.24** Programa para repartir las cartas. (Parte 3 de 3.)

```
 Nueve de corazones       Cinco de treboles
 Quina de espadas          Tres de espadas
 Quina de corazones          As de treboles
   Rey de corazones        Seis de espadas
  Joto de diamantes       Cinco de espadas
 Siete de corazones         Rey de treboles
  Tres de treboles         Ocho de corazones
  Tres de diamantes      Cuatro de diamantes
 Quina de diamantes       Cinco de diamantes
  Seis de diamantes       Cinco de corazones
    As de espadas          Seis de corazones
 Nueve de diamantes       Quina de treboles
  Ocho de espadas         Nueve de treboles
   Dos de treboles         Seis de treboles
   Dos de espadas          Joto de treboles
Cuatro de treboles         Ocho de treboles
Cuatro de espadas         Siete de espadas
 Siete de diamantes       Siete de treboles
   Rey de espadas          Diez de diamantes
  Ocho de diamantes         Dos de diamantes
    As de diamantes       Nueve de espadas
Cuatro de corazones         Dos de corazones
   Rey de diamantes        Diez de espadas
  Tres de corazones        Diez de corazones
```

**Figura 7.25** Muestra de la ejecución del programa para repartir las cartas.

Existe una debilidad en el algoritmo para repartir. Una vez que se encuentra una coincidencia, incluso si se encuentra en el primer intento, las dos instrucciones **for** internas continúan la búsqueda en los elementos restantes de **mazo** por una coincidencia. Corregiremos esta deficiencia en los ejercicios y en el ejemplo práctico del capítulo 10.

## 7.12 Apuntadores a funciones

Un *apuntador a una función* contiene la dirección de la función en memoria. En el capítulo 6, vimos que el nombre de un arreglo es en realidad la dirección en memoria del primer elemento del arreglo. De manera similar, el nombre de una función es en realidad la dirección inicial en memoria del código que realiza la tarea de la función. Los apuntadores a funciones pueden pasarse a funciones, ser devueltos desde funciones, ser almacenados en arreglos y asignados a otros apuntadores a funciones.

Para ilustrar el uso de los apuntadores a funciones, en la figura 7.26 presentamos una versión modificada del programa de ordenamiento de burbuja de la figura 7.15. La nueva versión consta de la función **main** y de las funciones **burbuja**, **intercambia**, **ascendente** y **descendente**. La función **ordenaMBurbuja** recibe como argumento un apuntador a una función, ya sea la función **ascendente** o la función **descendente**, además del arreglo entero y el tamaño de éste. El programa indica al usuario que elija si el arreglo debe ordenarse de manera ascendente o descendente. Si el usuario escribe **1**, se pasa el apuntador a la función

**ascendente** hacia la función **burbuja**, lo que provoca que el arreglo sea ordenado en orden creciente. Si el usuario escribe 2, se pasa el apuntador a la función **descendente** hacia la función **burbuja**, lo que provoca que el arreglo sea ordenado en orden decreciente. La salida de programa aparece en la figura 7.27.

```
/* Figura 7.26: fig07_26.c
   Programa de ordenamiento multipropósito que utiliza apuntadores a funciones */
#include <stdio.h>
#define TAMANIO 10

/* prototipos */
void burbuja( int trabajo[], const int tamanio, int (*compara)( int a, int b ) );
int ascendente( int a, int b );
int descendente( int a, int b );

int main()
{
   int orden;     /* 1 para el orden ascendente o 2 para el orden descendente */
   int contador; /* contador */

   /* inicializa el arreglo a */
   int a[ TAMANIO ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( “Introduzca 1 para ordenar en forma ascendente,\n”
           “Introduzca 2 para ordenar en forma descendente: “ );
   scanf( “%d”,  &orden );

   printf( “\nElementos de datos en el orden original\n” );

   /* muestra el arreglo original */
   for ( contador = 0; contador < TAMANIO; contador++ ) {
      printf( “%5d”, a[ contador ] );
   } /* fin de for */

   /* clasifica el arreglo en orden ascendente; pasa la función ascendente como un
   argumento para especificar el orden ascendente */
   if ( orden == 1 ) {
      burbuja( a, TAMANIO, ascendente );
      printf( “\nElementos de datos en orden ascendente\n” );
   } /* fin de if */
   else { /* pasa la función descendente */
      burbuja( a, TAMANIO, descendente );
      printf( “\nElementos de datos en orden descendente\n” );
   } /* fin de else */

   /* muestra el arreglo ordenado */
   for ( contador = 0; contador < TAMANIO; contador++ ) {
      printf( “%5d”, a[ contador ] );
   } /* fin de for */

   printf( “\n” );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 7.26** Programa de ordenamiento multipropósito con apuntadores a funciones. (Parte 1 de 2.)

```
/* ordenamiento burbuja multipropósito; el parámetro compara es un apuntador a
   la función de comparación que determina el tipo de ordenamiento */
void burbuja( int trabajo[], const int tamanio, int (*compara)( int a, int b ) )
{
   int pasada;  /* contador de pasadas */
   int cuenta;  /* contador de comparaciones */

   void intercambia( int *ptrElemento1, int *ptrElemento2 ); /* prototipo */

   /* ciclo para controlar las pasadas */
   for ( pasada = 1; pasada < tamanio; pasada++ ) {

      /* ciclo para controlar el número de comparaciones por pasada */
      for ( cuenta = 0; cuenta < tamanio - 1; cuenta++ ) {

         /* si los elementos adyacentes no se encuentran en orden,
            los intercambia */
         if ( (*compara)( trabajo[ cuenta ], trabajo[ cuenta + 1 ] ) ) {
            intercambia( &trabajo[ cuenta ], &trabajo[ cuenta + 1 ] );
         } /* fin de if */

      } /* fin de for */

   } /* fin de for */

} /* fin de la función burbuja */

/* intercambia los valores en las ubicaciones de memoria a las que apunta
   ptrElemento1 y ptrElemento2 */
void intercambia( int *ptrElemento1, int *ptrElemento2 )
{
   int almacena; /* variable de almacenamiento temporal */

   almacena = *ptrElemento1;
   *ptrElemento1 = *ptrElemento2;
   *ptrElemento2 = almacena;
} /* fin de la función intercambia */

/* determina si los elementos están en desorden para un
   ordenamiento ascendente */
int ascendente( int a, int b )
{
   return b < a;   /* intercambia si b es menor que a */

} /* fin de la función ascendente */

/* determina si los elementos están en desorden para un
   ordenamiento descendente */
int descendente( int a, int b )
{
   return b > a;   /* intercambia si b es mayor que a */

} /* fin de la función descendente */
```

**Figura 7.26** Programa de ordenamiento multipropósito con apuntadores a funciones. (Parte 2 de 2.)

```
Introduzca 1 para ordenar en forma ascendente,
Introduzca 2 para ordenar en forma descendente: 1

Elementos de datos en el orden original
    2    6    4    8   10   12   89   68   45   37
Elementos de datos en orden ascendente
    2    4    6    8   10   12   37   45   68   89
```

```
Introduzca 1 para ordenar en forma ascendente,
Introduzca 2 para ordenar en forma descendente: 2

Elementos de datos en el orden original
    2    6    4    8   10   12   89   68   45   37
Elementos de datos en orden descendente
   89   68   45   37   12   10    8    6    4    2
```

**Figura 7.27** Las salidas del programa de ordenamiento multipropósito de la figura 7.26.

El siguiente parámetro aparece en el encabezado de la función burbuja (línea 54)

```
int (*compara)( int a, int b )
```

Esto indica a **burbuja** que espere un parámetro (**compara**) que es un apuntador a una función que recibe dos parámetros enteros y que devuelva un resultado entero. Los paréntesis son necesarios alrededor de ***compara**, para agrupar a * con **compara** y para indicar que **compara** es un apuntador. Si no incluimos el paréntesis, la declaración podría ser

```
int *compara( int a, int b )
```

la cual declara una función que recibe dos enteros como parámetros y devuelve un apuntador a un entero.

El prototipo de función para **burbuja** aparece en la línea 7. Observe que el prototipo podría escribirse como

```
int (*)( int, int );
```

sin el nombre del apuntador a la función, ni los nombres de los parámetros.

La función que se pasa a **burbuja** se llama en una instrucción **if** (línea 68) como sigue

```
if ((*compara)(trabaja[ cuenta ], trabaja[ cuenta + 1 ] ) )
```

Tal como un apuntador a una variable se desreferencia para acceder el valor de la variable, un apuntador a una función se desreferencia para utilizar la función.

La llamada a la función se podría haber hecho sin desreferenciar el apuntador como en

```
if ( compara( trabaja[ cuenta ], trabaja[ cuenta + 1 ] ) )
```

la cual utiliza un apuntador directamente hacia el nombre de la función. Preferimos el primer método para llamar a una función a través de un apuntador, debido a que explica de manera explícita que **compara** es un apuntador a una función que se desreferencia para llamar a una función. El segundo método para llamar a una función a través de un apuntador lo hace aparecer como si **compara** fuera en realidad una función. Esto puede ser confuso para un usuario del programa que quiere ver la definición de la función **compara** y encuentra que no existe tal definición dentro del archivo.

***Cómo utilizar apuntadores a funciones para crear un sistema basado en menús***

Un uso común de los *apuntadores a funciones* se encuentra en los llamados sistemas basados en menús. A un usuario se le indica que seleccione una opción desde un menú (posiblemente de 1 a 5). Cada opción se sirve de una función diferente. Los apuntadores a cada función se almacenan en un arreglo de apuntadores a funciones. Las opciones del usuario se utilizan como subíndices del arreglo, y el apuntador en el arreglo se utiliza para llamar a una función.

La figura 7.28 proporciona un ejemplo genérico de la mecánica para definir y utilizar un arreglo de apuntadores a funciones. Se definen tres funciones, **funcion1**, **funcion2** y **funcion3**, las cuales toman un argumento entero y no devuelven valor alguno. Los apuntadores a estas tres funciones se almacenan en el arreglo **f**, el cual se define de la siguiente manera (línea 14):

```
void ( *f[ 3 ] )( int ) = { funcion1, funcion2, funcion3 };
```

La definición se lee desde el paréntesis que se encuentra hasta la izquierda, “**f** es un arreglo de 3 apuntadores a funciones que toman un **int** como argumento y que devuelven **void**”. El arreglo se inicializa con los nombres de las tres funciones. Cuando el usuario introduce un valor entre 0 y 2, el valor se utiliza como el subíndice del arreglo de apuntadores a funciones. La llamada a la función (línea 26) se hace de la siguiente manera:

```
(*f[ eleccion ])( eleccion );
```

```
1  /* Figura 7.28: fig07_28.c
2     Demostración de un arreglo de apuntadores a funciones */
3  #include <stdio.h>
4
5  /* prototipos */
6  void funcion1( int a );
7  void funcion2( int b );
8  void funcion3( int c );
9
10 int main()
11 {
12    /* inicializa el arreglo de 3 apuntadores con funciones que toman
13       un argumento entero y devuelven void */
14    void (*f[ 3 ])( int ) = { funcion1, funcion2, funcion3 };
15
16    int eleccion; /* variable para almacenar la elección del usuario */
17
18    printf( “Introduzca un numero entre 0 y 2, 3 para terminar: “ );
19    scanf( “%d”, &eleccion );
20
21    /* procesa la elección del usuario */
22    while ( eleccion >= 0 && eleccion < 3 ) {
23
24       /* invoca a la función en la ubicación de la elección en el arreglo f, y pasa
25          la elección como argumento */
26       (*f[ eleccion ])( eleccion );
27
28       printf( “Introduzca un numero entre 0 y 2, 3 para terminar: “);
29       scanf( “%d”, &eleccion );
30    } /* fin de while */
31
32    printf( “Termina le ejecucion del programa.\n” );
33
34    return 0; /* indica terminación exitosa */
35
36 } /* fin de main */
37
38 void funcion1( int a )
39 {
40    printf( “Usted introdujo %d de manera que invoco a la funcion1\n\n”, a );
```

**Figura 7.28** Demostración de un arreglo de apuntadores a funciones. (Parte 1 de 2.)

```
} /* fin de la funcion1 */

void funcion2( int b )
{
   printf( "Usted introdujo %d de manera que invoco a la funcion2\n\n", b );
} /* fin de la funcion2 */

void funcion3( int c )
{
   printf( "Usted introdujo %d de manera que invoco a la funcion2\n\n", c );
} /* fin de la funcion3 */
```

```
Introduzca un numero entre 0 y 2, 3 para terminar: 0
Usted introdujo 0 de manera que invoco a la funcion1

Introduzca un numero entre 0 y 2, 3 para terminar: 1
Usted introdujo 1 de manera que invoco a la funcion2

Introduzca un numero entre 0 y 2, 3 para terminar: 2
Usted introdujo 2 de manera que invoco a la funcion2

Introduzca un numero entre 0 y 2, 3 para terminar: 3
Termina la ejecucion del programa.
```

**Figura 7.28** Demostración de un arreglo de apuntadores a funciones. (Parte 2 de 2.)

En la llamada de la función, **`f[ eleccion ]`** selecciona el apuntador que se encuentra en la ubicación **`eleccion`** del arreglo. El apuntador se desreferencia para llamar a la función y **`eleccion`** se pasa como el argumento de la función. Cada función imprime el valor de su argumento y su nombre de función para demostrar que la función se invoca correctamente. En los ejercicios, usted desarrollará un sistema basado en menús.

## RESUMEN

- Los apuntadores son variables que contienen como sus valores las direcciones de otras variables.
- Los apuntadores deben definirse antes de utilizarlos.
- La definición

  **`int *ptr;`**

  define a **`ptr`** como un apuntador a un objeto de tipo **`int`** y se lee, "**`ptr`** es un apuntador a un **`int`**". Aquí, el **`*`** se utiliza para indicar que la variable es un apuntador.
- Existen tres valores que pueden utilizarse para inicializar un apuntador: **`0`**, **`NULL`**, o una dirección. Inicializar un apuntador en **`0`**, o inicializar el mismo apuntador en **`NULL`** es lo mismo.
- El único entero que puede asignarse a un apuntador es **`0`**.
- El operador de dirección (**`&`**) devuelve la dirección del operando.
- El operando del operador de dirección debe ser una variable; el operador de dirección no puede aplicarse a constantes, expresiones, o a variables declaradas con la clase de almacenamiento **`register`**.
- El operador **`*`**, conocido como operador de indirección o desreferencia, devuelve el valor de memoria del objeto al cual apunta su operando. A esto se le llama desreferenciar un apuntador.
- Cuando llamamos a una función con un argumento que queremos que la función modifique, pasamos la dirección del argumento. Después, la función que se invoca utiliza el operador de indirección (**`*`**) para modificar el valor del argumento de la función que se invocó.
- Una función que recibe una dirección como argumento debe incluir un apuntador a su parámetro formal correspondiente.

- No es necesario incluir los nombres de los apuntadores en el prototipo de la función; sólo es necesario incluir el tipo de los apuntadores. Los nombres de los apuntadores pueden incluirse por razones de documentación, pero el compilador los ignora.
- El calificador **`const`** permite al programador informar al compilador que no se puede modificar el valor de una variable en particular.
- Si se intenta modificar un valor declarado como **`const`**, el compilador lo atrapa y despliega un mensaje de error o de advertencia, dependiendo del compilador en particular.
- Existen cuatro maneras de pasar un apuntador a una función: un apuntador no constante a un dato no constante, un apuntador constante a un dato no constante, un apuntador no constante a un dato constante y un apuntador constante a un dato constante.
- Los arreglos se pasan por referencia de manera automática, debido a que el valor del nombre del arreglo es la dirección del mismo arreglo.
- Para pasar por referencia un solo elemento de un arreglo a una función, debe pasarse la dirección específica del elemento del arreglo.
- C proporciona el operador unario especial **`sizeof`** para determinar el tamaño en bytes de un arreglo (o cualquier otro tipo de dato), en tiempo de compilación.
- Cuando se aplica el operador **`sizeof`** al nombre de un arreglo, éste devuelve un entero que representa el número total de bytes del arreglo.
- El operador **`sizeof`** puede aplicarse a cualquier nombre de variable, tipo o constante.
- El tipo **`size_t`** es un tipo definido en el encabezado (**`<stddef.h>`**) como el tipo integral (**`unsigned`** o **`unsigned long`**) del valor devuelto por el operador **`sizeof`**.
- Las operaciones aritméticas que pueden aplicarse a los apuntadores son: incremento de un apuntador (**`++`**), decremento de un apuntador (**`--`**), suma (**`+`** o **`+=`**) de un apuntador y un entero, resta (**`-`** o **`-=`**) de un apuntador a un entero, y la resta de un apuntado a otro.
- Cuando se suma o se resta un entero a un apuntador, éste se incrementa o decrementa el número de veces enteras del tamaño del objeto al cual apunta.
- Las operaciones aritméticas con apuntadores sólo pueden realizarse en porciones contiguas de memoria, tales como arreglos. Todos los elementos de un arreglo se almacenan en espacios contiguos de memoria.
- Cuando se aplica la aritmética de apuntadores sobre un arreglo de carácteres, los resultados son como en la aritmética normal, debido a que cada carácter se almacena en un byte de memoria.
- Los apuntadores pueden asignarse uno a otro, si ambos son del mismo tipo. La excepción a esto es un apuntador a **`void`**, el cual es un tipo genérico de apuntador que puede apuntar a datos de cualquier tipo. A los apuntadores a **`void`** se les pueden asignar apuntadores de otros tipos y pueden asignarse a apuntadores de otros tipos sin una conversión.
- No se debe desreferenciar un apuntador a **`void`**.
- Los apuntadores pueden compararse por medio de los operadores de igualdad y de relación. Por lo general, la comparación de apuntadores es valiosa sólo si apuntan a miembros del mismo arreglo.
- A los apuntadores se les puede asignar subíndices de la misma manera que a los nombres de arreglos.
- Un nombre de arreglo sin un subíndice es un apuntador al primer elemento del arreglo.
- En la notación apuntador/desplazamiento, el desplazamiento hace lo mismo que el subíndice de un arreglo.
- Todas las expresiones con arreglos con subíndices pueden escribirse por medio de un apuntador y un desplazamiento, por medio del mismo nombre de arreglo como un apuntador, o por medio de un apuntador separado que apunta al arreglo.
- El nombre de un arreglo es un apuntador constante que apunta siempre a la misma posición de memoria. Los nombres de arreglo no pueden modificarse como los apuntadores.
- Es posible tener arreglos de apuntadores.
- Es posible tener apuntadores a funciones.
- Un apuntador a una función es la dirección en donde reside el código de la función.
- Los apuntadores a funciones pueden pasarse como funciones, devolverse como funciones, almacenarse en arreglos y asignarse a otros apuntadores.
- Un uso común de los apuntadores a funciones es en los llamados sistemas basados en menús.

## TERMINOLOGÍA

aplazamiento indefinido
apuntador
apuntador a un carácter
apuntador a una función
apuntador a **void(void *)**
apuntador constante
apuntador constante a un dato constante
apuntador constante a un dato no constante
apuntador de función
apuntador no constante a un dato constante
apuntador no constante a un dato no constante
apuntador **NULL**
aritmética de apuntadores
arreglo de apuntadores
arreglo de cadenas
asignación de apuntadores
asignación dinámica de memoria
comparación de apuntadores
**const**
decremento de un apuntador
desplazamiento
desreferencia de un apuntador
expresión con apuntadores
incremento de un apuntador
indexación de apuntadores
indirección
inicialización de apuntadores
lista ligada
llamada por referencia
llamada por valor
notación apuntador/desplazamiento
operador de desreferencia (*****)
operador de dirección (**&**)
operador de indirección (*****)
operador **sizeof**
principio del menor privilegio
referencia directa a una variable
referencia indirecta a una variable
refinamiento arriba-abajo, paso a paso
resta de dos apuntadores
resta de un entero de un apuntador
simulación de una llamada por referencia
subíndices de apuntadores
suma de un apuntador y un entero
tipo **size_t**
tipos de apuntadores
**void *** (apuntador a **void**)

## ERRORES COMUNES DE PROGRAMACIÓN

**7.1** La notación asterisco (*****) que se utiliza para declarar variables de tipo apuntador no se distribuye a todas las variables en la declaración. Cada apuntador debe declararse con el prefijo ***** en el nombre, por ejemplo, si desea declarar **ptrX** y **ptrY** como apuntadores int, utilice **int *ptrX**, ***ptrY**;

**7.2** Desreferenciar un apuntador que no se inicializó de manera apropiada, o que no se le indicó que apunte hacia una dirección específica en memoria es un error. Esto podría provocar un error fatal en tiempo de ejecución, o podría modificar de manera accidental datos importantes y permitir la ejecución del programa pero con resultados incorrectos.

**7.3** No desreferenciar un apuntador cuando es necesario hacerlo para obtener el valor al que apunta el apuntador, es un error de sintaxis.

**7.4** No estar consciente de que una función espera apuntadores como argumentos para realizar una llamada por referencia y para pasar argumentos por valor. Algunos compiladores toman los valores y asumen que son apuntadores, por lo que desreferencian los valores como apuntadores. A tiempo de ejecución, a menudo generan violaciones de acceso a memoria o fallas de segmentación. Otros compiladores atrapan el error de tipos entre los argumentos y los parámetros, y generan mensajes de error.

**7.5** Utilizar la aritmética de apuntadores sobre un apuntador que no hace referencia a un elemento de un arreglo.

**7.6** Restar o comparar dos apuntadores que no hacen referencia a los elementos del mismo arreglo.

**7.7** Rebasar el final de un arreglo cuando se utiliza la aritmética de apuntadores.

**7.8** Asignar un apuntador de un tipo específico a un apuntador de otro tipo, incluso si es de tipo **void ***, es un error de sintaxis.

**7.9** Desreferenciar un apuntador **void ***, es un error de sintaxis.

**7.10** Intentar modificar el nombre del arreglo con aritmética de apuntadores, es un error de sintaxis.

## TIPS PARA PREVENIR ERRORES

**7.1** Inicialice los apuntadores para prevenir resultados inesperados.

**7.2** Utilice llamadas por valor para pasar argumentos a una función, a menos que la función que hace la llamada requiera explícitamente que la función que se invoca modifique el valor del argumento en el entorno de la función que hace la llamada. Esto previene modificaciones accidentales de los argumentos en la llamada de la función, y es otro ejemplo del principio del menor privilegio.

**7.3** Si una variable no se modifica (o no debiera modificarse) en el cuerpo de la función a la que se pasa, la variable debe declararse como **`const`** para garantizar que no se modifique de manera accidental.

**7.4** Antes de usar una función, verifique su prototipo para determinar si la función es capaz de modificar los valores que se le pasan.

## *BUENA PRÁCTICA DE PROGRAMACIÓN*

**7.1** Incluya las letras **`ptr`** en los nombres de las variables de apuntadores para hacer más claro que estas variables son apuntadores y, por lo tanto, que deben manipularse de manera apropiada.

## *TIPS DE RENDIMIENTO*

**7.1** El paso de objetos grandes, tales como estructuras, utilizando apuntadores a datos constantes, obtiene las ventajas de una llamada por referencia y la seguridad de una llamada por valor.

**7.2** Pasar el tamaño de un arreglo a una función toma tiempo y requiere espacio adicional en la pila, debido a que se crea una copia del tamaño para pasarla a la función. Las variables globales no requieren tiempo o espacio adicional, debido a que cualquier función puede acceder a ellas de manera directa.

**7.3** **`sizeof`** es un operador en tiempo de compilación, de manera que no implica sobrecarga alguna en tiempo de ejecución.

**7.4** Algunas veces un algoritmo que emerge de manera "natural" puede contener sutiles problemas de rendimiento, tales como el aplazamiento indefinido. Busque algoritmos que eviten el aplazamiento indefinido.

## *TIPS DE PORTABILIDAD*

**7.1** Aunque **`const`** está bien definido en el ANSI C, algunos compiladores no lo soportan.

**7.2** El número de bytes que se utilizan para almacenar un tipo de dato en particular puede variar entre sistemas. Cuando escriba programas que dependan del tamaño del tipo de dato y que se ejecutarán en varios sistemas de computadoras, utilice **`sizeof`** para determinar el número de bytes requeridos para almacenar los tipos de datos.

**7.3** La mayoría de las computadoras actuales tienen enteros de 2 y 4 bytes. Algunas de las máquinas más nuevas utilizan enteros de 8 bytes. Debido a que los resultados de la aritmética de apuntadores dependen del tamaño de los objetos al que apunta el apuntador, la aritmética de apuntadores depende de la máquina.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**7.1** El calificador **`const`** puede utilizarse para reforzar el principio del menor privilegio. Utilizar el principio del menor privilegio para diseñar software de manera apropiada, reduce el tiempo de depuración y los efectos colaterales indeseados, lo que hace a un programa más fácil de modificar y de mantener.

**7.2** Sólo se puede alterar un valor en la función invocada cuado utilizamos una llamada por referencia. El valor debe asignarse desde el valor de retorno de la función. Para modificar valores en la función invocada, debe utilizar una llamada por referencia.

**7.3** Colocar los prototipos de las funciones en la definición de otras funciones promueve el principio del menor privilegio, al restringir las llamadas a las funciones, a aquellas en donde aparece su prototipo.

**7.4** Cuando pase un arreglo a una función, también pase el tamaño del arreglo. Esto ayuda a hacer a la función reutilizable en muchos programas.

**7.5** A menudo, las variables globales violan el principio del menor privilegio y pueden provocar una pobre ingeniería de software.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**7.1** Responda cada una de las siguientes preguntas:

a) Una variable de apuntador contiene como su valor la __________ de otra variable.

b) Los tres valores que pueden utilizarse para inicializar un apuntador son ___________, __________ o _________.

c) El único entero que puede asignarse a un apuntador es el ___________.

**7.2** Diga si los siguientes enunciados son *verdaderos o falsos*. Si la respuesta es *falso,* explique por qué.

a) El operador de dirección (**&**)puede aplicarse sólo a constantes, a expresiones y a variables declaradas con la clase de almacenamiento **`register`**.

b) Un apuntador declarado como **`void`** se puede desreferenciar.

c) Los apuntadores con tipos diferentes no pueden asignarse entre sí, sin un operador de conversión de tipo.

**7.3** Responda a cada una de las siguientes preguntas. Suponga que los números de punto flotante de precisión simple se almacenan en 4 bytes de memoria, y que la dirección inicial del arreglo es la ubicación de memoria 1002500. Cada parte del ejercicio debe utilizar los resultados de las partes previas, en donde sea apropiado.

a) Defina un arreglo de tipo **`float`** llamado **`numeros`** con 10 elementos, e inicialice los elementos con los valores **`0.0, 1.1, 2.2..., 9.9.`** Suponga que la constante simbólica **`TAMANIO`** se definió como **`10`**.

b) Defina un apuntador, **`ptrN`**, que apunte a un objeto de tipo **`float`**.

c) Imprima los elementos del arreglo **`numeros`** mediante la notación de subíndices. Utilice una instrucción **`for`** y suponga que la variable entera de control **`i`** ya se definió. Imprima cada número con **`1`** posición de precisión a la derecha del punto decimal.

d) Escriba dos instrucciones separadas que asignen la dirección inicial del arreglo **`numeros`** a la variable de apuntador **`ptrN`**.

e) Imprima los elementos del arreglo **`numeros`** mediante la notación apuntador/desplazamiento con el apuntador **`ptrN`**.

f) Imprima los elementos del arreglo **`numeros`** mediante la notación apuntador/desplazamiento con el nombre del arreglo como apuntador.

g) Imprima los elementos del arreglo **`numeros`** colocando un subíndice al apuntador **`ptrN`**.

h) Haga referencia al elemento 4 del arreglo **`numeros`** mediante la notación de arreglos con subíndices, la notación apuntador/desplazamiento con el nombre del arreglo como apuntador, la notación de arreglos con subíndices con **`ptrN`** y la notación apuntador/desplazamiento con **`ptrN`**.

i) Suponga que **`ptrN`** apunta al inicio del arreglo **`numeros`**, ¿A cuál dirección se hace referencia con **`ptrN + 8`**? ¿Cuál valor se almacena en dicha ubicación?

j) Suponga que ptrN apunta a **`numeros[ 5 ]`**, ¿a cuál dirección se hace referencia mediante **`ptrN -= 4`**? ¿Cuál es el valor almacenado en dicha ubicación?

**7.4** Para cada uno de los siguientes enunciados, escriba una instrucción que realice la tarea indicada. Suponga que las variables de punto flotante **`numero1`** y **`numero2`** ya se definieron, y que **`numero1`** se inicializa en **`7.3`**.

a) Defina la variable **`ptrF`** como un apuntador a un objeto de tipo **`float`**.

b) Asigne la dirección de la variable **`numero1`** hacia el apuntador **`ptrF`**.

c) Imprima el valor del objeto al que apunta **`ptrF`**.

d) Asigne a la variable **`numero2`** el valor del objeto al que apunta **`ptrF`**.

e) Imprima el valor de **`numero2`**.

f) Imprima la dirección de **`numero1`**. Utilice el especificador de conversión **`%p`**.

g) Imprima la dirección almacenada en **`ptrF`**. Utilice el especificador de conversión **`%p`**. ¿El valor impreso es el mismo que el de **`numero1`**?

**7.5** Realice cada una de las siguientes actividades:

a) Escriba el encabezado para la función llamada **`intercambio`**, la cual toma como parámetros a dos apuntadores a los números de punto flotante **`x`** y **`y`**, y no devuelve valor alguno.

b) Escriba el prototipo de función para la función de la parte (a).

c) Escriba un encabezado para la función llamada **`evalua`**, la cual devuelve un entero y toma como parámetros los números enteros **`x`** y el apuntador a la función **`poli`**. La función **`poli`** toma un parámetro entero y devuelve un entero.

d) Escriba el prototipo de función para la función del inciso (c).

**7.6** Encuentre el error en cada uno de los segmentos de programa. Suponga que

```
int *ptrZ; /* ptrZ hará referencia al arreglo z */
int *ptrA = NULL;
void *ptrS = NULL;
int numero, i ;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
ptrS = z;
```

a) `++ptrZ;`
b) `/* utiliza el apuntador para obtener el valor del primer elemento del arreglo */`
```
numero = ptrZ;
```
c) `/* asigna el elemento 2 del arreglo (el valor 3) a número */`
```
numero = *ptrZ[ 2 ];
```
d) `/* imprime el arreglo Z completo */`
```
for ( i = 0; i <= 5; i++ )
    printf( "%d ", ptrZ[ i ] );
```
e) `/* asigna a numero el valor al que apunta ptrS */`
```
numero = *ptrS;
```
f) `++z;`

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**7.1** a) Dirección. b) 0, `NULL`, una dirección. c) `0`.

**7.2** a) Falso. El operador de dirección sólo puede aplicarse a variables. El operador de dirección no puede aplicarse a variables que se declaran con la clase de almacenamiento `register`.
b) Falso. No se puede desreferenciar un apuntador a `void`, debido a que no hay forma de saber con exactitud cuántos bytes de memoria desreferenciar.
c) Falso. A los apuntadores de tipo `void` se les pueden asignar apuntadores de otros tipos, y los apuntadores de tipo `void` pueden asignarse a apuntadores de otros tipos.

**7.3** a)
```
float numeros[ TAMANIO ] =
      { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
```
b) `float *ptrN;`
c)
```
for ( i = 0; i < TAMANIO; i++ )
      printf( "%.1f ", numeros[ i ]);
```
d)
```
ptrN = numeros;
ptrN = &numeros[ 0 ];
```
e)
```
for ( i = 0; i < TAMANIO; i++ )
      printf( "%.1f ", *(ptrN + i ) );
```
f)
```
for ( i = 0; i < TAMANIO; i++ )
      printf( "%.1f ", *(numeros + i ) );
```
g)
```
for ( i = 0; i < TAMANIO; i++ )
      printf( "%.1f ", ptrN[ i ] );
```
h)
```
numeros[ 4 ]
*( numeros + 4 )
ptrN[ 4 ]
*( ptrN + 4 )
```
i) La dirección es `1002500 + 8 * 4 = 1002532.` El valor es `8.8`.
j) La dirección de `numeros[ 5 ]` es `1002500+5*4=1002520`.
La dirección de `ptrN -=4 es 1002520-4*4=1002504`.
El valor de esa ubicación es `1.1`.

**7.4** a) `float *ptrF;`
b) `ptrF = &numero1;`
c) `printf( "El valor de *ptrF es %f\n", *ptrF );`
d) `numero2 = *ptrF;`
e) `printf( "El valor de numero2 es %f\n", numero2 );`
f) `printf( "La dirección de numero1 es %p\n", &numero1 );`
g) `printf( "La dirección almacenada en ptrF es %p\n", ptrF );`
Sí, el valor es el mismo.

**7.5** a) `void intercambio( float *x, float *y )`
b) `void intercambio( float *x, float *y );`
c) `int evalua( int x, int (*poli)( int ) )`
d) `int evalua( int x, int (*poli)( int ) );`

**7.6** a) Error: **`ptrZ`** no se inicializó.
Corrección: inicialice **`ptrZ`** como **`ptrZ=z;`**
b) Error: no se desreferenció el apuntador.
Corrección: cambie la instrucción por **`numero=*ptrZ;`**
c) Error: **`ptrZ[ 2 ]`** no es un apuntador y por lo tanto no se debe desreferenciar.
Corrección: cambie **`*ptrZ[ 2 ]`** por **`ptrZ[ 2 ]`**.
d) Error: está haciendo referencia a un elemento del arreglo fuera de los límites de éste, por medio de subíndices de apuntador.
Corrección: modifique el operador **`<=`** por **`<`**, en la condición de la instrucción **`for`**.
e) Error: desreferenciar a un apuntador **`void`**.
Corrección: para poder desreferenciar al apuntador, primero se debe convertir a un apuntador entero. **`Modifique`** la instrucción a **`numero = *( ( int * ) ptrS );`**
f) Error: intenta modificar un nombre de arreglo mediante la aritmética de apuntadores.
Corrección: utilice una variable de apuntador, en lugar del nombre de un arreglo, para llevar a cabo la aritmética de apuntadores, o coloque subíndices al nombre del arreglo para hacer referencia al elemento específico.

## *EJERCICIOS*

**7.7** Responda a cada una de las siguientes preguntas:
a) El operador ___________ devuelve la ubicación en memoria donde se almacena su operando.
b) El operador ___________ devuelve el valor del objeto al cual apunta su operando.
c) Para simular una llamada por referencia cuando pasamos a la función una variable que no es un arreglo, es necesario pasar a la función la ________ de la variable.

**7.8** Diga si los enunciados siguientes son *verdaderos* o *falsos*. Si son *falsos*, explique por qué.
a) Dos apuntadores que apuntan hacia arreglos diferentes no pueden compararse de manera significativa.
b) Debido a que el nombre de un arreglo es un apuntador al primer elemento del mismo arreglo, los nombres de arreglos deben manipularse precisamente de la misma manera que los apuntadores.

**7.9** Responda cada una de las siguientes preguntas. Suponga que los enteros unsigned se almacenan en 2 bytes y que la dirección inicial del arreglo en memoria es la **`1002500`**.
a) Defina un arreglo de tipo **`unsigned int`**, con cinco elementos, llamado **`valores`**, e inicialice los elementos en los cinco enteros pares de **`2`** a **`10`**. Suponga que la constante simbólica **`TAMANIO`** se definió como **`5`**.
b) Defina el apuntador **`ptrY`** para que apunte a objetos de tipo **`unsigned int`**.
c) Imprima los elementos del arreglo **`valores`** mediante la notación de subíndices para arreglos. Use una instrucción **`for`** y suponga que ya se definió la variable de control entera **`i`**.
d) Escriba dos instrucciones separadas para asignar la dirección inicial del arreglo **`valores`** a la variable de apuntador **`ptrV`**.
e) Imprima lo elementos del arreglo **`valores`** mediante la notación apuntador/desplazamiento.
f) Imprima los elementos del arreglo **`valores`** mediante el uso de la notación apuntador/desplazamiento con el nombre del arreglo como apuntador.
g) Imprima los elementos del arreglo **`valores`** mediante subíndices en el apuntador al arreglo.
h) Haga referencia al elemento **`5`** del arreglo **`valores`** mediante el uso de la notación de subíndices, apuntador/desplazamiento con el nombre del arreglo como apuntador, notación de subíndices de apuntadores, y notación apuntador/desplazamiento.
i) ¿A qué dirección hace referencia **`ptrV + 3`**? ¿Qué valor se almacena en dicha ubicación?
j) Suponga que **`ptrV`** apunta a **`valores[ 4 ]`**, ¿a qué dirección hace referencia **`ptrV -= 4`**? ¿Cuál es el valor que se almacena en dicha ubicación?

**7.10** Para cada una de las siguientes, escriba una sola instrucción que realice la tarea indicada. Suponga que se definieron las variables **`long`** integer **`valor1`** y **`valor2`**, y que **`valor1`** se inicializó en **`200000`**.
a) Defina la variable **`ptrL`** para que apunte a un objeto de tipo **`long`**.
b) Asigne la dirección de la variable **`valor1`** para que apunte a la variable **`ptrL`**.
c) Imprima el valor del objeto al que apunta **`ptrL`**.
d) Asigne a la variable **`valor2`** el valor del objeto al que apunta **`ptrL`**.
e) Imprima el valor de **`valor2`**.
f) Imprima la dirección de **`valor1`**.
g) Imprima la dirección almacenada en **`ptrL`**. ¿El valor que se imprimió es el mismo que la dirección de **`valor1`**?

**7.11** Realice cada una de las siguientes acciones.
a) Escriba el encabezado de la función **`cero`**, la cual toma como parámetro el arreglo de enteros largos **`enterosGrandes`** y no devuelve valor alguno.
b) Escriba el prototipo para la función del *inciso a*.
c) Escriba el encabezado de la función para **`agrega1ySuma`**, la cual toma como parámetro el arreglo de enteros **`unoApequeno`** y devuelve un entero.
d) Escriba el prototipo para la función del *inciso c*.

***Nota: los ejercicios 7.12 a 7.15 son relativamente complejos. Una vez que haya hecho estos problemas, será capaz de implementar los juegos de cartas más populares de manera sencilla.***

**7.12** Modifique el programa de la figura 7.24 de manera que la función para repartir las cartas reparta una mano de póquer de cinco cartas. Después, escriba las siguientes funciones adicionales:
a) Determine si la mano contiene un par.
b) Determine si la mano contiene dos pares.
c) Determine si la mano contiene tres de un solo tipo (por ejemplo, tres jotos).
d) Determine si la mano contiene cuatro del mismo tipo (por ejemplo, cuatro ases).
e) Determine si la mano contiene las cinco cartas del mismo palo.
f) Determine si la mano contiene una directa (es decir, cinco cartas del mismo palo y con caras consecutivas).

**7.13** Utilice las funciones desarrolladas en el ejercicio 7.12 para escribir un programa que reparta dos manos de póquer de cinco cartas, evalúe cada mano, y determine cuál es la mejor mano.

**7.14** Modifique el programa desarrollado en el ejercicio 7.13 de manera que pueda simular al repartidor. La mano de cinco cartas del repartidor se da con la “cara abajo”, de manera que el jugador no las puede ver. Entonces, el programa debe evaluar la mano del repartidor, y basado en la calidad de la mano, el repartidor debe tirar una, dos o más cartas y remplazar el número de cartas tiradas en la mano original. Después, el programa debe reevaluar la mano del repartidor. [*Precaución:* ¡Éste es un problema difícil!]

**7.15** Modifique el programa desarrollado en el ejercicio 7.14 de manera que pueda manipular la mano del repartidor de manera automática, pero al jugador se le debe permitir decidir cuáles cartas desea remplazar. Entonces, el programa debe evaluar ambas manos y determinar quién gana. Utilice el nuevo programa para jugar 20 juegos contra la computadora. ¿Quién gana más juegos? Basado en los resultados de estos juegos, haga las modificaciones apropiadas para redefinir el programa de póquer (esto también es un problema difícil), Juegue 20 juegos más. ¿Sus modificaciones hicieron que su programa funcionara mejor?

**7.16** En el programa para barajar y repartir cartas de la figura 7.24, utilizamos de manera intencional un algoritmo ineficiente que tiene la posibilidad latente de un aplazamiento indefinido. En este problema, usted creará un algoritmo de alto rendimiento para barajar cartas que evite el aplazamiento indefinido.

Modifique el programa de la figura 7.24 de la siguiente manera. Comience mediante la inicialización del **`mazo`** de cartas como lo mostramos en la figura 7.29. Modifique la función **`barajar`** para hacer un ciclo que explore línea por línea y columna por columna para tocar todos loes elementos del arreglo. Cada elemento debe intercambiarse con el elemento del arreglo seleccionado al azar.

**Arreglo mazo no barajado**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 1 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 2 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 3 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |

**Figura 7.29** Arreglo **`mazo`** no barajado.

**Arreglo mazo barajado**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 19 | 40 | 27 | 25 | 36 | 46 | 10 | 34 | 35 | 41 | 18 | 2 | 44 |
| 1 | 13 | 28 | 14 | 16 | 21 | 30 | 8 | 11 | 31 | 17 | 24 | 7 | 1 |
| 2 | 12 | 33 | 15 | 42 | 43 | 23 | 45 | 3 | 29 | 32 | 4 | 47 | 26 |
| 3 | 50 | 38 | 52 | 39 | 48 | 51 | 9 | 5 | 37 | 49 | 22 | 6 | 20 |

**Figura 7.30** Arreglo **`mazo`** barajado.

Imprima el arreglo resultante para determinar si el mazo se barajó de manera satisfactoria (por ejemplo, como en la figura 7.30). Usted puede llamar a la función **`barajar`** varias veces para asegurarse que el mazo se barajó de manera satisfactoria.

Observe que aunque el método en este problema mejora el algoritmo para barajar las cartas, el algoritmo para repartir requiere la búsqueda del arreglo **`mazo`** para la carta1, carta2, carata3, carta4, y así sucesivamente. Peor aún, incluso cuando el algoritmo para repartir localiza y maneja las cartas, el algoritmo continúa la búsqueda a través del resto del **`mazo`**. Modifique el programa de la figura 7.24 de manera que una vez que la carta se reparte, no se hagan más intentos para hacer coincidir el número de la carta, y que el programa proceda de inmediato a repartir la siguiente carta. En el capítulo 10, desarrollaremos un algoritmo para repartir que requiere solamente una operación por carta.

**7.17** (*Simulación: la tortuga y la liebre.*) En este problema, usted recreará uno de los grandes momentos de la historia, a saber, la clásica carrera entre la tortuga y la liebre. Usted utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores, comienzan la carrera en la “posición 1” de 70. La línea final se encuentra en la posición 70. Al primer competidor en alcanzar o pasar el cuadrante 70 se le recompensará con un montón de zanahorias frescas y lechuga. La ruta va a lo largo de una sinuosa montaña, de manera que ocasionalmente los competidores se caerán.

Existe un reloj que hace un tic por segundo. Con cada tic del reloj, su programa debe ajustar la posición de los animales de acuerdo con las reglas de la figura 7.31.

Utilice variables para dar seguimiento a las posiciones de los animales (es decir, los números de las posiciones entre 1 y 70). Comience cada animal en la posición 1 (es decir, la “puerta inicial”). Si un animal se desliza a la izquierda antes de la posición 1, mueva al animal de nuevo a la posición 1.

Genere los porcentajes en la tabla anterior mediante la producción de un entero aleatorio, i, en el rango de $1 \leq i \leq 10$ . Para la tortuga, realice un “paso rápido” cuando $1 \leq i \leq 5$, un “deslizamiento” cuando $6 \leq i \leq 7$, o un “paso lento cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Para comenzar la carrera imprima

```
BANG !!!!!
Y ARRANCAN !!!!!
```

| Animal | Tipo de movimiento | Porcentaje del tiempo | Movimiento real |
|---|---|---|---|
| Tortuga | Paso rápido | 50% | 3 posiciones a la derecha |
| | Deslizamiento | 20% | 6 posiciones a la izquierda |
| | Paso lento | 30% | 1 posición a la derecha |
| Liebre | Dormir | 20% | Sin movimiento |
| | Salto grande | 20% | 9 posiciones a la derecha |
| | Deslizamiento grande | 10% | 12 posiciones a la izquierda |
| | Salto pequeño | 30% | 1 posición a la derecha |
| | Deslizamiento pequeño | 20% | 2 posiciones a la izquierda |

**Figura 7.31** Reglas para ajustar las posiciones de la tortuga y la liebre.

Posteriormente, por cada tic del reloj (es decir, cada repetición del ciclo), imprima una línea de 70 posiciones que muestre la letra T en la posición de la tortuga y una letra L en la posición de la liebre. Ocasionalmente, los competidores caerán en la misma posición. En este caso, la tortuga muerde a la liebre y su programa debe imprimir un **`OUCH!!!!!`**, comenzando en dicha posición. Todas las posiciones además de la T, y de la L, o de **`OUCH!!!!!`** (en caso de un empate) deben estar en blanco.

Después de que se imprima una línea, verifique si el animal ya alcanzó o pasó la posición 70. Si es así, entonces imprima el nombre del ganador y termine la simulación. Si la tortuga gana, imprima **`GANO LA TORUTUGA!!!!! VIVA !!!!!`**. Si gana la liebre, imprima **`Gano la Liebre`**. **`Yupi`**. Si ambos animales ganan en el mismo tic del reloj, usted puede favorecer a la tortuga (por "debajo del agua"), o puede imprimir **`Es un empate`**. Si ningún animal gana la carrera, ejecute de nuevo el ciclo para simular el siguiente tic del reloj. Cuando esté preparado para ejecutar su programa, reúna a un grupo de amigos para que vea la carrera. ¡Usted se sorprenderá por la manera en que su público se involucra!

## *SECCIÓN ESPECIAL: CONSTRUYA SU PROPIA COMPUTADORA*

En los próximos problemas, nos alejaremos un poco de los lenguajes de programación de alto nivel. "Abriremos" una computadora y examinaremos su estructura interna. Nos introduciremos al lenguaje máquina y escribiremos varios programas en dicho lenguaje. Para hacer de esto una experiencia significativa, construiremos (a través de la técnica de la simulación basada en *software*) una computadora en la cual usted podrá ejecutar sus programas en lenguaje máquina.

**7.18** (*Programación en lenguaje máquina.*) Vamos a crear una computadora a la cual llamaremos Simpletron. Como su nombre lo indica, es una máquina simple, pero como veremos pronto, también es poderosa. El Simpletron ejecuta programas escritos en el único lenguaje que comprende de manera directa, esto es el Lenguaje Máquina de Simpletron, LMS.

El Simpletron contiene un *acumulador*, un "registro especial" en el cual, la información se coloca antes de que el Simpletron utilice dicha información en los cálculos o que la examine de distintas manera. Toda la información en el Simpletron se maneja mediante *palabras*. Una palabra es un número decimal de cuatro dígitos con signo, tal como **`+3364, -1293, +0007, -0001`**, etcétera. El Simpletron está equipado con **`100`** palabras de memoria, y se hace referencia a estas palabras mediante su número de ubicación **`00, 01, …, 99`**.

Antes de ejecutar un programa LMS, debemos *cargarlo* o colocarlo dentro de la memoria. La primera instrucción de cada programa LMS siempre se coloca en la ubicación **`00`**.

Cada instrucción escrita en LMS ocupa una palabra en la memoria del Simpletron (y por lo tanto, las instrucciones son números decimales de cuatro dígitos). Asumimos que el signo de una instrucción LMS siempre es positivo, pero el signo de una palabra de datos puede ser positivo o negativo. Cada dirección en la memoria del Simpletron puede contener una instrucción, un valor de dato que el programa utiliza o un área de memoria sin utilizar (por lo tanto, indefinida). Los primeros dos dígitos de cada instrucción LMS representan el código de operación, el cual especifica la operación a realizar. Resumimos los códigos de operación de LMS en la figura 7.32.

| Código de operación | Significado |
|---|---|
| *Operaciones de entrada/salida*: | |
| **`#define LEE 10`** | Lee una palabra desde la terminal y la almacena en la ubicación de memoria. |
| **`#define ESCRIBE 11`** | Escribe una palabra desde una ubicación específica de memoria hacia la terminal. |
| *Operaciones de carga/almacenamiento* | |
| **`#define CARGA 20`** | Carga una palabra desde la ubicación específica de memoria hacia el acumulador. |
| **`#define ALMACENA 21`** | Almacena una palabra desde el acumulador hacia una ubicación específica de memoria. |
| *Operaciones aritméticas*: | |
| **`#define SUMA 30`** | Suma una palabra desde una ubicación específica de memoria a la palabra almacenada en el acumulador (deja el resultado en el acumulador). |

**Figura 7.32** Códigos de operación del Lenguaje máquina Simpletron (LMS). (Parte 1 de 2.)

| Código de operación | Significado |
|---|---|
| `#define RESTA 31` | Resta una palabra desde una ubicación específica de memoria del acumulador (deja el resultado en el acumulador). |
| `#define DIVIDE 32` | Divide una palabra desde una ubicación específica de memoria entre la palabra dentro del acumulador (deja el resultado en el acumulador). |
| `#define MULTIPLICA 33` | Multiplica una palabra desde una ubicación específica de memoria por la palabra almacenada dentro del acumulador (deja el resultado en el acumulador). |
| *Operaciones de transferencia de control:* | |
| `#define SALTA 40` | Salta a una ubicación específica de memoria. |
| `#define SALTANEG 41` | Salta hacia una ubicación específica de memoria si el acumulador es negativo. |
| `#define SALTACERO 42` | Salta a una ubicación específica de memoria si el acumulador es igual a cero. |
| `#define ALTO 43` | Para, es decir, el programa finalizó su tarea. |

**Figura 7.32** Códigos de operación del Lenguaje máquina Simpletron (LMS). (Parte 2 de 2.)

Los dos últimos dígitos de una instrucción LMS son el *operando*, el cual es la dirección de la ubicación de memoria que contiene la palabra a la cual se aplica la operación. Ahora, consideremos varios ejemplos de programas en LMS:

| Ejemplo 1<br>Ubicación | Número | Instrucción |
|---|---|---|
| **00** | **+1007** | (*Lee A*) |
| **01** | **+1008** | (*Lee B*) |
| **02** | **+2007** | (*Carga A*) |
| **03** | **+3008** | (*Suma B*) |
| **04** | **+2109** | (*Almacena C*) |
| **05** | **+1109** | (*Escriba C*) |
| **06** | **+4300** | (*Alto*) |
| **07** | **+0000** | (*Variable A*) |
| **08** | **+0000** | (*Variable B*) |
| **09** | **+0000** | (*Resultado C*) |

El programa LMS anterior lee dos números desde el teclado, y calcula e imprime la suma. La instrucción **+1007** lee el primer número desde el teclado y lo coloca dentro de la ubicación **07** (la cual se inicializa en cero). Posteriormente, **+1008** lee el siguiente número en la ubicación **08**. La instrucción *Carga*, **+2007**, coloca el primer número dentro del acumulador, y la instrucción suma, **+3008**, suma el segundo número al número que se encuentra en el acumulador. *Todas las instrucciones LMS dejan los resultados dentro del acumulador*. La instrucción *almacena*, **+2109**, coloca el resultado en la ubicación de memoria **09** desde la cual, la instrucción *escribe*, **+1109**, toma el número y lo imprime (como un número de cuatro dígitos decimales con signo). La instrucción *alto*, **+4300** termina la ejecución.

| Ejemplo 2<br>Ubicación | Número | Instrucción |
|---|---|---|
| **00** | **+1009** | (*Lee A*) |
| **01** | **+1010** | (*Lee B*) |
| **02** | **+2009** | (*Carga A*) |
| **03** | **+3110** | (*Resta B*) |
| **04** | **+4107** | (*Salta a 07 si acumulador es negativo*) |

| Ejemplo 2 Ubicación | Número | Instrucción |
|---|---|---|
| **05** | **+1109** | (*Escribe A*) |
| **06** | **+4300** | (*Alto*) |
| **07** | **+1110** | (*Escribe B*) |
| **08** | **+4300** | (*Alto*) |
| **09** | **+0000** | (*Variable A*) |
| **10** | **+0000** | (*Variable B*) |

El programa LMS anterior lee dos números desde el teclado, y determina e imprime el valor más grande. Observe el uso de la instrucción **+4107** como la transferencia condicional de control, muy parecida a la instrucción **if** de C. Ahora escriba programas LMS para llevar a cabo cada una de las siguientes tareas.

a) Utilice un ciclo controlado por centinela para leer **10** enteros positivos y calcular e imprimir la suma.
b) Utilice un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcule e imprima su promedio.
c) Lea una serie de números y determine e imprima el número más grande. El primer número leído indica cuántos números se deben procesar.

**7.19** (*Un simulador de computadora.*) Podría parecer descabellado, pero en este problema usted va a construir su propia computadora. No, no va a soldar los componentes. En vez de ello, utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* del Simpletron. No se decepcionará. Su simulador del Simpletron convertirá a la computadora que usted utiliza en un Simpletron, y en realidad será capaz de ejecutar, probar y corregir los programas LMS que escribió en el ejercicio 7.18.

Cuando ejecute su propio simulador de Simpletron, éste debe comenzar con la impresión de:

```
*** Bienvenido a Simpletron!                          ***
*** Por favor, introduzca a su programa una instruccion ***
*** a la vez (o palabra de datos).          ***
*** Yo escribire el numero de ubicacion y un      ***
*** signo de interrogacion (?). Usted escriba ***
*** la palabra para dicha ubicacion. Escriba el   ***
*** centinela -99999 para terminar la    ***
*** introduccion de datos a su programa.                    ***
```

Simule la memoria del Simpletron mediante un arreglo con un solo subíndice llamado **memoria**, con 100 elementos. Ahora suponga la ejecución del simulador, y permita que examinemos el diálogo mientras introducimos el programa del ejemplo 2 del ejercicio 7.18.

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Carga del programa completa ***
*** Comienza la ejecucion del programa ***
```

El programa LMS se encuentra ahora dentro del arreglo de memoria. Ahora, Simpletron ejecutará su programa. La ejecución comienza con la instrucción en la ubicación **00** y, como en C, continúa de manera secuencial, a menos que la dirijamos a otra parte del programa mediante una transferencia de control.

Utilice la variable **acumulador** para representar el registro del acumulador. Utilice la variable **contadorInstrucciones** para llevar el registro de la ubicación en memoria que contiene a la instrucción que se ejecuta. Utilice la variable **coigoOperacion** para indicar la operación que se va a realizar, es decir, los dos dígitos a la izquierda de la palabra de instrucción. Utilice la variable **operando** para indicar la ubicación de memoria en la cual opera la instrucción actual. Además, **operando** son los dos dígitos a la derecha de la instrucción que se

encuentra en ejecución. No ejecute instrucciones de manera directa desde la memoria. En vez de esto, transfiera la siguiente instrucción a ejecutarse desde la memoria hacia la variable llamada **`registroInstruccion`**. Después "tome" los dos dígitos a la izquierda y colóquelos dentro de la variable **`codigoOperacion`**, y "tome" los dos dígitos de la derecha y colóquelos dentro de **`operando`**.

Cuando comienza la ejecución de Simpletron, se inicializan los registros especiales de la siguiente manera:

```
acumulador                  +0000
contadorInstrucciones          00
registroInstruccion         +0000
codigoOperacion                00
operando                       00
```

Ahora, recorramos la ejecución de la primera instrucción de LMS, **`+1009`** en la ubicación de memoria **`00`**. A esto le llamamos *ciclo de ejecución de la instrucción*.

El **`contadorInstrucciones`** nos indica la ubicación la siguiente instrucción a ejecutarse. *Extraemos* el contenido de dicha ubicación de **`memoria`** mediante la instrucción de C

```
registroIntruccion = memoria[ contadorInstrucciones ];
```

El código de operación y el operando se extraen desde el registro de instrucciones mediante las instrucciones

```
codigoOperacion = registroInstruccion / 100;
operando = registroInstruccion % 100;
```

Ahora, el Simpletron debe determinar si el código de operación es en realidad un *lee* (*versus* un *escribe*, un *carga*, etcétera). Un **`switch`** diferencia entre las veinte operaciones de LMS.

En la instrucción **`switch`**, el comportamiento de las distintas instrucciones LMS se simulan de la siguiente manera (dejamos las demás al lector):

*lee*: **`scanf( "%d", &memoria[ operando ] );`**

*carga*: **`acumulador = memoria [ operando ];`**

*suma*: **`acumulador += memoria [ operando ];`**

*Las distintas instrucciones para saltos: las explicaremos más adelante.*

*alto*: Esta instrucción imprime el mensaje

```
*** Finaliza ejecución de Simpletron ***
```

entonces imprime el nombre y el contenido de cada registro así como el contenido completo de la memoria. A menudo, a tal impresión se le llama *vaciado de la computadora*. Para ayudarle a programar su función de vaciado,

```
REGISTROS:
acumulador                 +0000
contadorInstrucciones         00
registroInstruccion        +0000
codigoOperacion               00
operando                      00

MEMORIA
        0      1      2      3      4      5      6      7      8      9
 0  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
10  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
20  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
30  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
40  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
50  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
60  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
70  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
80  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
90  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
```

**Figura 7.33** Ejemplo de vaciado de memoria de Simpletron.

mostramos un formato de ejemplo en la figura 7.33. Observe que la ejecución del programa Simpletron mostrará los valores actuales de las instrucciones y los valores de los datos al momento de la terminación de la ejecución.

Procedamos con la ejecución de la primera instrucción del programa, a saber, `+1009` en la ubicación `00`. Como lo indicamos, la instrucción `switch` simula esto mediante la instrucción

```
scanf( "%d", &memoria[ operando ] );
```

Se debe desplegar un signo de interrogación (`?`) en la pantalla para indicar al usuario la entrada, antes de que se ejecute la instrucción `scanf`. Simpletron espera que el usuario escriba el valor y presione la tecla de *Retorno*. Entonces, se lee el valor en la ubicación `09`.

En este punto, termina la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Ya que la instrucción que se ejecutó no era una transferencia de control, sólo necesitamos incrementar el contador de instrucciones de la siguiente manera:

```
++contadorInstrucciones;
```

Esto competa la simulación de la ejecución de la primera instrucción. El proceso completo (es decir, el ciclo de ejecución de la instrucción) comienza con la extracción de la siguiente instrucción que se va a ejecutar.

Ahora examinemos cómo se simulan las instrucciones de salto (transferencia de control). Todo lo que tenemos que hacer es ajustar el valor del contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de salto no condicional (40) se simula dentro de `switch` como

```
contadorInstrucciones = operando;
```

La instrucción condicional "salta si acumulador es cero" se simula como

```
if ( acumulador  == 0 )
   contadorInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar los programas que escribió en el ejercicio 7.18. Puede embellecer el LMS con características adicionales y proporcionarlas a su simulador.

Su simulador debe evaluar distintos tipos de errores. Por ejemplo, durante la carga del programa, cada número que escribe el usuario dentro de la `memoria` de Simpletron debe estar en el rango de `-9999` a `+9999`. Su simulador debe utilizar un ciclo `while` para evaluar que cada número introducido se encuentre en este rango, y si no, indique al usuario que rescriba el número hasta que el usuario introduzca el número correcto.

Durante la fase de ejecución, su simulador debe verificar distintos errores fatales, tales como intentos de dividir entre cero, intentos de ejecutar códigos inválidos de operación y desbordamientos del acumulador (es decir, operaciones aritméticas que resulten en valores mayores a `+9999` o menores a `-9999`). Tales errores se llaman *errores fatales*. Cuando se detecta un error fatal, su simulador debe imprimir un mensaje de error como el siguiente:

```
*** Intento de division entre cero ***
*** Terminacion anormal del programa ***
```

y debe imprimir un vaciado completo de memoria mediante el formato que explicamos anteriormente. Esto ayudará al usuario a localizar el error en el programa.

**7.20** Modifique el programa para barajar y repartir cartas de la figura 7.24, de manera que las operaciones de barajar y repartir se realicen dentro de la misma función (`barajarYrepartir`). La función debe contener una estructura de ciclo anidada, similar a la función `barajar` de la figura 7.24.

**7.21** ¿Qué hace el siguiente programa?

```
/* ej07_21.c */
/* ¿Qué hace este programa? */
#include <stdio.h>

void misterio1( char *s1, const char *s2 ); /* prototipo */

int main()
{
```

(Parte 1 de 2.)

```
   char cadena1[ 80 ]; /* crea un arreglo de caracteres */
   char cadena2[ 80 ]; /* crea un arreglo de caracteres */

   printf( “Introduce dos cadenas: “ );
   scanf( “%s%s” , cadena1, cadena2 );

   misterio1( cadena1, cadena2 );

   printf(“%s”, cadena1 );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* ¿Qué hace esta función ? */
void misterio1( char *s1, const char *s2 )
{
   while ( *s1 != ‘\0’ ) {
      s1++;
   } /* fin de while */

   for ( ; *s1 = *s2; s1++, s2++ ) {
      ;   /* instrucción vacía */
   } /* fin de for */

} /* fin de la función misterio1 */
```

(Parte 2 de 2.)

**7.22** ¿Qué hace el siguiente programa?

```
/* ej07_22.c */
/* ¿Qué hace este programa? */
#include <stdio.h>

int misterio2( const char *s ); /* prototipo */

int main()
{
   char cadena[ 80 ]; /* crea un arreglo de carateres */

   printf( “Introduzca una cadena: “);
   scanf( “%s”, cadena );

   printf( “%d\n”, misterio2( cadena ) );

   return 0; /* indica terminación exitosa */
} /* fin de main */

/* ¿Qué hace esta función? */
int misterio2( const char *s )
{
   int x; /* contador */

   /* ciclo a través de la cadena */
```

(Parte 1 de 2.)

```
    for ( x = 0; *s != ‘\0’; s++ ) {
       x++;
    } /* fin de for */

    return x;

} /* fin de la función misterio2 */
```

(Parte 2 de 2.)

**7.23** Encuentre el error en cada una de las siguientes porciones de programa. Si se puede corregir el error, explique cómo:

a)
```
int *numero;
printf( “%d\n”, *numero );
```
b)
```
float *ptrReal;
long *ptrEntero;
ptrEntero = ptrReal;
```
c)
```
int * x, y;
x = y;
```
d)
```
char s[] = “este es un arreglo de caracteres”;
int cuenta;
for ( ; *s != ‘\0’; s++)
   printf( “%c”, *s );
```
e)
```
short *ptrNum, resultado;
void *ptrGenerico = ptrNum;
resultado = *ptrGenerico + 7;
```
f)
```
float x = 19.34;
float ptrX = &x;
printf( “%f\n”, ptrX );
```
g)
```
char *s;
printf( “%s\n”, s );
```

**7.24** (*Quicksort.*) En los ejemplos y los ejercicios del capítulo 6, explicamos las técnicas de ordenamiento por los métodos de burbuja, cubetas y selección. Ahora explicaremos la técnica recursiva de ordenamiento llamada Quicksort.

El algoritmo básico para los valores de un arreglo con un solo subíndice es el siguiente:

a) *Paso para particionar*. Tome el primer elemento del arreglo desordenado y determine su ubicación final en el arreglo clasificado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores que el elemento). Ahora, tenemos el elemento en su ubicación principal y dos subarreglos desordenados.

b) *Paso recursivo*. Realiza el *paso 1* en cada subarreglo desordenado.
Cada vez que se realiza el *paso 1* en un subarreglo, se coloca otro elemento en su ubicación final dentro del arreglo ordenado, y se crean dos arreglos desordenados. Cuando un subarreglo consiste de un solo elemento, éste debe clasificarse; por lo tanto, dicho elemento se encuentra en su ubicación final.

El algoritmo básico parece bastante sencillo, ¿pero cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el elemento para la partición, éste se colocará en su ubicación final en el arreglo ordenado):

**37** 2 6 4 89 8 10 12 68 45

a) El proceso comienza por el elemento que se encuentra a la extrema derecha de arreglo, y compara cada elemento con **37** hasta que encuentra un elemento menor. Entonces intercambia **37** con ese elemento. El primer elemento menor a **37** es 12, de manera que **37** y 12 se intercambian. El nuevo arreglo es

*12* 2 6 4 89 8 10 **37** 68 45

El elemento *12* está en cursivas para indicar que acaba de intercambiarse con **37**.

b) Comenzando desde la izquierda del arreglo, pero después del elemento *12*, compara cada elemento con **37** hasta encontrar un elemento mayor. Entonces, intercambia **37** y el elemento. El primer elemento mayor que **37** es 89, de manera que **37** y 89 se intercambian. El nuevo arreglo es

12 2 6 4 **37** 8 10 *89* 68 45

c) Comenzando desde la derecha, pero antes del elemento 89, compara cada elemento con **37** hasta encontrar un elemento menor. Entonces, intercambia **37** y el elemento. El primer elemento menor que **37** es 10, de manera que **37** y 10 se intercambian. El nuevo arreglo es

12 2 6 4 *10* 8 **37** 89 68 45

d) Comenzando desde la izquierda, pero después del elemento 10, compara cada elemento con **37** hasta encontrar un elemento mayor. Entonces, intercambia **37** y el elemento. Ya no existen elementos mayores que **37**, entonces al compararse con sí mismo, sabemos que **37** se encuentra en su posición final en el arreglo ordenado.

Una vez que se aplica la partición al arreglo, existen dos arreglos desordenados. El subarreglo con valores menores que 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con los valores mayores que 37 contienen 89, 68 y 45. El ordenamiento continúa con la partición de ambos arreglos de la misma manera que en el arreglo original.

Escriba la función recursiva **`quicksort`** para ordenar un arreglo con un solo subíndice. La función debe recibir como argumentos un arreglo de enteros, un subíndice de inicio y un subíndice final. La función partición debe invocarse mediante **`quicksort`** para realizar el paso para la partición.

**7.25** (*Recorrido de laberintos.*) La siguiente rejilla es arreglo con dos subíndices que representa un laberinto.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . .
# . . . . # # # . # . #
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Los símbolos # representan las paredes del laberinto, y los puntos (**.**) representan posiciones de la posible ruta a través del laberinto.

Existe un algoritmo sencillo para recorrer los laberintos, que garantiza el poder encontrar la salida (asumiendo que existe una salida). Si no existe salida, usted llegará de nuevo a la ubicación inicial. Coloque su mano derecha en la pared y comience a caminar hacia delante. Nunca despegue su mano de la pared, seguirá la pared hacia la derecha. Mientras usted no mueva su mano de la pared, tarde o temprano llegará a la salida del laberinto. Podría existir una ruta más corta que la que usted tomó, pero esto le garantiza la salida del laberinto.

Escriba una función recursiva **`recorreLaberinto`** para recorrer el laberinto. La función debe recibir como argumentos un arreglo de caracteres de 12 por 12, que represente al laberinto y a la ubicación inicial del laberinto. Mientras **`recorreLaberinto`** intenta localizar la salida del laberinto, debe colocar el carácter **`X`** en cada posición del arreglo en la ruta. La función debe desplegar el laberinto después de cada movimiento, de manera que el usuario pueda observar cómo se resuelve el laberinto.

**7.26** (*Generación de laberintos al azar.*) Escriba una función **`generadorLaberintos`** que tome como argumento un arreglo de caracteres de 12 por 12 elementos y que produzca laberintos de manera aleatoria. Además, la función debe proporcionar las posiciones inicial y final del laberinto. Pruebe su función **`recorreLaberinto`** del ejercicio 7.25, utilizando laberintos generados al azar.

**7.27** (*Laberintos de cualquier tamaño.*) Generalice las funciones **`recorreLaberinto`** y **`generadorLaberintos`** de los ejercicios 7.25 y 7.26 para procesar laberintos de cualquier ancho y alto.

**7.28** (*Arreglos de apuntadores a funciones.*) Rescriba el programa de la figura 6.22 para utilizar una interfaz basada en menús. El programa debe ofrecer las cuatro opciones que aparecen a continuación:

```
Elija una opcion:
  0   Imprime el arreglo de calificaciones
  1   Encuentra la calificacion minima
  2   Encuentra la calificacion maxima
  3   Imprime el promedio de todos los examenes de cada estudiante.
  4   Fin del programa
```

Una restricción en el uso de arreglos de apuntadores a funciones es que todos los apuntadores deben tener el mismo tipo. Los apuntadores deben ser hacia funciones con el mismo tipo de retorno y que reciban argumentos del mismo tipo. Por esta razón, deben modificarse las funciones de la figura 6.22 de manera que cada una devuelva el mismo tipo y tome los mismos parámetros. Modifique las funciones **minimo** y **maximo** para imprimir el mínimo o máximo valor, y que no devuelva valor alguno. Para la opción 3, modifique la función **promedio** de la figura 6.22 para desplegar el promedio de cada estudiante (no un estudiante en especial). La función **promedio** no debe devolver valor alguno y debe tomar los mismos parámetros que **imprimeArreglo**, **minimo** y **maximo**. Almacene todos los apuntadores en las cuatro funciones dentro del arreglo **procesaCalificaciones** y utilice la opción elegida por el usuario como el subíndice dentro del arreglo para llamar a cada función.

**7.29** (*Modificaciones al simulador de Simpletron.*) En el ejercicio 7.19, usted escribió una simulación basada en el software de una computadora que ejecuta programas escritos en Lenguaje Máquina de Simpletron (LMS). En este ejercicio, proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 12.26 y 12.27, proponemos la construcción de un compilador que convierta programas escritos en un lenguaje de programación de alto nivel (una variación de BASIC) a Lenguaje Máquina de Simpletron. Éstas son algunas de las modificaciones y mejoras que se podrían requerir para ejecutar programas producidos por el compilador.

a) Extienda la memoria del simulador Simpletron para que contenga 1000 direcciones de memoria y así permitir al Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de residuos. Esto requiere una instrucción adicional en el lenguaje Máquina de Simpletron.
c) Permita al simulador realizar cálculos de exponenciación. Esto requiere una instrucción adicional en el Lenguaje Máquina de Simpletron.
d) Modifique el simulador para utilizar valores hexadecimales, en lugar de valores enteros para representar instrucciones en Lenguaje Máquina de Simpletron.
e) Modifique el simulador para permitir la salida de una línea nueva. Esto requiere una instrucción adicional del Lenguaje Máquina de Simpletron.
f) Modifique el simulador para poder procesar valores de punto flotante, además de los valores enteros.
g) Modifique el simulador para manejar entrada y salida de cadenas. [*Pista:* Cada palabra en Simpletron puede dividirse en dos grupos, cada uno almacena un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente en ASCII decimal de un carácter. Agregue una instrucción en lenguaje máquina que introduzca un carácter y la almacene el principio de la cadena en una ubicación específica de la memoria de Simpletron. La primera mitad de la palabra en dicha ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y la asigna a la mitad de la palabra.]
h) Modifique el simulador para manipular la salida de cadenas almacenadas en el formato del inciso (g). [*Pista:* Agregue una instrucción en lenguaje máquina que imprima el principio de una cadena en una ubicación específica de la memoria de Simpletron. La primera mitad de la palabra en dicha ubicación es la longitud de la cadena en caracteres. Cada mitad de palabra subsiguiente contiene un carácter ASCII representado como dos dígitos decimales. La instrucción en lenguaje máquina verifica la longitud e imprime la cadena mediante la traducción de cada número de dos dígitos en su carácter equivalente.]

**7.30** ¿Qué hace este programa?

```
/* ej07_30.c */
/* ¿Qué hace este programa? */
#include <stdio.h>

int misterio3( const char *s1, const char *s2 ); /* prototipo */

int main()
{
   char cadena1[ 80 ]; /* crea un arreglo de caracteres */
   char cadena2[ 80 ]; /* crea un arreglo de caracteres */

   printf( "Introduzca dos cadenas: " );
   scanf( "%s%s", cadena1 , cadena2 );
```

(Parte 1 de 2.)

```
   printf( "El resultado es %d\n", misterio3( cadena1, cadena2 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */

int misterio3( const char *s1, const char *s2 )
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ ) {

    if ( *s1 != *s2 ) {
          return 0;
      } /* fin de if */

   } /* fin de for */

   return 1;

} /* fin de la función misterio3 */
```

(Parte 2 de 2.)



# Caracteres y cadenas en C

## Objetivos

- Utilizar funciones de la biblioteca de manipulación de caracteres (**`ctype`**).
- Utilizar funciones de entrada/salida de caracteres y cadenas de la biblioteca estándar de entrada/salida (**`stdio`**).
- Utilizar funciones de conversión de cadenas de la biblioteca general de utilidades (**`stdlib`**).
- Utilizar funciones para procesamiento de cadenas de la biblioteca de manipulación de cadenas (**`string`**).
- Apreciar el poder de las bibliotecas de funciones como medio para lograr la reutilización de software.




*El principal defecto del rey Enrique era que masticaba pequeños trozos de hilo.*
Hilaire Belloc

*Empata la acción con la palabra, que la palabra sea acción.*
William Shakespeare

*La escritura vigorosa es concisa. Una oración no debe contener palabras innecesarias, y un párrafo no debe contener oraciones innecesarias.*
William Strunk, Jr.

*De acuerdo con la concatenación.*
Oliver Goldsmith

## Plan general



## 8.1 Introducción

En este capítulo, presentamos las funciones de la biblioteca estándar que facilitan el procesamiento de cadenas y caracteres. Las funciones permiten a los programas procesar caracteres, cadenas, líneas de texto y bloques de memoria.

El capítulo explica las técnicas empleadas para desarrollar editores, procesadores de palabras, software de diseño de páginas, sistemas de captura computarizada y otro tipo de software de procesamiento de texto. Las manipulaciones de texto realizadas por las funciones de entrada/salida con formato como **`printf`** y **`scanf`** pueden implementarse mediante las funciones que explicamos en este capítulo.

## 8.2 Fundamentos de cadenas y caracteres

Los caracteres son los bloques de construcción fundamentales para los programas fuente. Cada programa está compuesto por una secuencia de caracteres que, cuando se agrupan apropiadamente, la computadora los interpreta como un conjunto de instrucciones utilizadas para llevar a cabo una tarea. Un programa puede contener *constantes de carácter*. Una constante de carácter es un valor **`int`** representado por un carácter entre comillas sencillas. El valor de una constante de carácter es el valor entero del carácter en el *conjunto de caracteres* de la máquina. Por ejemplo, **`'z'`** representa el valor entero de **`z`**, y **`'\n'`** representa el valor entero de una nueva línea.

Una *cadena* es un conjunto de caracteres tratados como una sola unidad. Una cadena puede incluir letras, dígitos y varios *caracteres especiales* como **`+`**, **`-`**, **`*`**, **`/`** y **`$`**. En C, las *literales de cadena*, o *constantes de cadena*, se escriben dentro de comillas dobles de la siguiente manera:

| | |
|---|---|
| "Juan P. Pérez" | (un nombre) |
| "99999 de Eje Central" | (la dirección de una calle) |
| "México, Distrito Federal" | (una ciudad y un estado) |
| "(55) 54 32 11 00" | (un número telefónico) |

En C, una cadena es un arreglo de caracteres, los cuales terminan con el *carácter nulo* (**`'\0'`**). Se accede a una cadena mediante un apuntador a su primer carácter. El valor de una cadena es la dirección del primer carácter. Así, en C, es apropiado decir que *una cadena es un apuntador*, de hecho, un apuntador al primer carácter de la cadena. En este sentido, las cadenas son como los arreglos, debido a que un arreglo también es un apuntador a su primer elemento.

Un arreglo de caracteres o una variable de tipo **char *** puede inicializarse con una cadena en la definición. Las definiciones

```
char color[] = "azul";
const char *ptrColor = "azul";
```

inicializan una variable con la cadena **"azul"**. La primera definición crea un arreglo de 5 elementos, **color**, que contiene los caracteres **'a'**, **'z'**, **'u'**, **'l'** y **'\0'**. La segunda definición crea una variable apuntador, **ptrColor**, que apunta a la cadena **"azul"** en algún lugar de la memoria.



### Tip de portabilidad 8.1

*Cuando se inicializa una variable de tipo* **char*** *con una literal de cadena, es posible que algunos compiladores coloquen la cadena en un lugar de la memoria, en donde ésta no se pueda modificar. Si necesitara modificar una literal de cadena, podría almacenarla en un arreglo de caracteres para garantizar que pueda modificarla en cualquier sistema.*

La definición del arreglo anterior también podría escribirse como

```
char color[] = { 'a', 'z', 'u', 'l', '\0' };
```

Cuando se define un arreglo para que contenga una cadena, éste debe ser lo suficiente grande para almacenar la cadena y su carácter de terminación nulo. La definición anterior determina automáticamente el tamaño del arreglo, basándose en el número de inicializadores de la lista de inicialización.



### Error común de programación 8.1

*No almacenar suficiente espacio en un arreglo de caracteres para almacenar el carácter nulo que termina una cadena, es un error.*



### Error común de programación 8.2

*Imprimir una "cadena" que no contiene el carácter de terminación nulo, es un error.*



### Tip para prevenir errores 8.1

*Cuando almacene una cadena de caracteres dentro de un arreglo, asegúrese de que el arreglo sea lo suficientemente grande para almacenar la cadena más larga que se vaya a guardar. C permite almacenar cadenas de cualquier longitud. Si una cadena es más grande que el arreglo de caracteres en el cual se va a almacenar, los caracteres más allá del final del arreglo sobrescribirán los datos siguientes en la memoria al arreglo.*

Una cadena puede almacenarse en un arreglo, por medio de **scanf**. Por ejemplo, la siguiente instrucción almacena el arreglo de caracteres **palabra[ 20 ]**:

```
scanf( "%s", palabra );
```

La cadena que introduce el usuario se almacena en **palabra**. Observe que **palabra** es un arreglo, el cual es, por supuesto, un apuntador, de modo que no necesitamos un **&** con el argumento **palabra**. La función **scanf** leerá caracteres hasta encontrar un espacio, un tabulador, un indicador de nueva línea o de fin de archivo. Observe que la cadena no debe ser mayor que 19 caracteres para dejar espacio suficiente para el carácter de terminación nulo. Para un arreglo de caracteres que se imprimirá como una cadena, el arreglo debe contener el carácter de terminación nulo.



### Error común de programación 8.3

*Procesar un solo carácter como una cadena. Una cadena es un apuntador, probablemente un entero de tamaño respetable. Sin embargo, un carácter es un entero pequeño (en el rango de valores ASCII 0-255). En muchos sistemas esto provoca un error, debido a que las direcciones de memoria baja se reservan para propósitos especiales tales como los manipuladores de interrupciones del sistema operativo, por lo que ocurren "violaciones de acceso".*



### Error común de programación 8.4

*Pasar un carácter como argumento a una función cuando se espera una cadena, es un error de sintaxis.*



### Error común de programación 8.5

*Pasar una cadena como un argumento a una función cuando se espera un carácter, es un error de sintaxis.*

## 8.3 La biblioteca de manipulación de caracteres

La *biblioteca de manipulación de caracteres* incluye varias funciones que realizan evaluaciones y manipulaciones útiles en datos de tipo carácter. Cada función recibe como argumento un carácter (representado como un **`int`**), o un **`EOF`**. Como explicamos en el capítulo 4, a menudo los caracteres se manipulan como enteros, debido a que por lo general en C, un carácter es un entero de 1 byte. En general, **`EOF`** contiene el valor −1 y en algunas arquitecturas de hardware no permiten almacenar valores negativos en las variables **`char`**, así, las funciones de manipulación de cadenas manipulan los caracteres como enteros. La figura 8.1 resume las funciones de la biblioteca de manipulación de caracteres.

| Prototipo | Descripción de la función |
|---|---|
| `int isdigit( int c );` | Devuelve un valor verdadero si **`c`** es un dígito; de lo contrario devuelve 0 (falso). |
| `int isalpha( int c );` | Devuelve un valor verdadero si **`c`** es una letra; de lo contrario devuelve 0 (falso). |
| `int isalnum( int c );` | Devuelve un valor verdadero si **`c`** es un dígito o una letra; de lo contrario devuelve 0 (falso). |
| `int isxdigit( int c );` | Devuelve un valor verdadero si **`c`** es un dígito hexadecimal; de lo contrario devuelve 0 (falso). (Revise el apéndice E, Sistemas de numeración, para una explicación detallada acerca de los números binarios, números octales, números decimales y números hexadecimales.) |
| `int islower( int c );` | Devuelve un valor verdadero si **`c`** es una letra minúscula; de lo contrario devuelve 0 (falso). |
| `int isupper( int c );` | Devuelve un valor verdadero si **`c`** es una letra mayúscula; de lo contrario devuelve 0 (falso). |
| `int tolower( int c );` | Si **`c`** es una letra mayúscula, **`tolower`** devuelve **`c`** como una letra minúscula. De lo contrario, **`tolower`** devuelve el argumento sin modificación. |
| `int toupper( int c );` | Si **`c`** es una letra minúscula, **`toupper`** devuelve **`c`** como una letra mayúscula. De lo contrario, toupper devuelve el argumento sin modificación. |
| `int isspace( int c );` | Devuelve un valor verdadero si **`c`** es un carácter de espacio en blanco (nueva línea (**`'\n'`**), espacio (**`' '`**), avance de página (**`'\f'`**), retorno de carro (**`'\r'`**), tabulador horizontal (**`'\t'`**) o tabulador vertical (**`'\v'`**); de lo contrario devuelve 0. |
| `int iscntrl( int c );` | Devuelve un valor verdadero si **`c`** es un carácter de control; de lo contrario devuelve 0 (falso). |
| `int ispunct( int c );` | Devuelve un valor verdadero si **`c`** es un carácter de impresión diferente de un espacio, un dígito o una letra; de lo contrario devuelve 0. |
| `int isprint( int c );` | Devuelve un valor verdadero si **`c`** es un carácter de impresión, incluso el espacio (**`' '`**); de lo contrario devuelve 0. |
| `int isgraph( int c );` | Devuelve un valor verdadero si **`c`** es un carácter de impresión diferente del espacio (**`' '`**); de lo contrario devuelve 0. |

**Figura 8.1** Funciones de la biblioteca de manipulación de caracteres.

### Tip para prevenir errores 8.2

*Cuando utilice funciones de la biblioteca de manipulación de caracteres, incluya el encabezado* **`<ctype.h>`**.

La figura 8.2 muestra las funciones ***`isdigit`***, ***`isalpha`***, ***`isalnum`*** e ***`isxdigit`***. La función **`isdigit`** determina si su argumento es un dígito (**`0-9`**). La función **`isalpha`** determina si su argumento es una letra mayúscula (**`A-Z`**), o una letra minúscula (**`a-z`**). La función **`isalnum`** determina si su argumento es una letra mayúscula, una letra minúscula o un dígito. La función **`isxdigit`** determina si su argumento es un *dígito hexadecimal* (**`A-F`**, **`a-f`**, **`0-9`**).

```
/* Figura 8.2: fig08_02.c
   Uso de las funciones isdigit, isalpha, isalnum, e isxdigit */
#include <stdio.h>
#include <ctype.h>

int main()
{
   printf( “%s\n%s%s\n%s%s\n\n”, “De acuerdo con isdigit: “,
       isdigit( ‘8’ ) ? “8 es un “ : “8 no es un “, “digito”,
       isdigit( ‘#’ ) ? “# es un “ : “# no es un “, “digito” );

   printf( “%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n”,
       “De acuerdo con isalpha:”,
       isalpha( ‘A’ ) ? “A es una “ : “A no es una “, “letra”,
       isalpha( ‘b’ ) ? “b es una “ : “b no es una “, “letra”,
       isalpha( ‘&’ ) ? “& es una “ : “& no es una “, “letra”,
       isalpha( ‘4’ ) ? “4 es una “ : “4 no es una “, “letra” );

   printf( “%s\n%s%s\n%s%s\n%s%s\n\n”,
       “De acuerdo con isalnum:”,
       isalnum( ‘A’ ) ? “A es un “ : “A no es un “,
       “digito o una letra”,
       isalnum( ‘8’ ) ? “8 es un “ : “8 no es un “,
       “digito o una letra”,
       isalnum( ‘#’ ) ? “# es un “ : “# no es un “,
       “digito o una letra” );

   printf( “%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n”,
       “De acuerdo con isxdigit:”,
       isxdigit( ‘F’ ) ? “F es un “ : “F no es un “,
       “digito hexadecimal”,
       isxdigit( ‘J’ ) ? “J es un “ : “J no es un “,
       “digito hexadecimal”,
       isxdigit( ‘7’ ) ? “7 es un “ : “7 no es un “,
       “digito hexadecimal”,
       isxdigit( ‘$’ ) ? “$ es un “ : “$ no es un “,
       “digito hexadecimal”,
       isxdigit( ‘f’ ) ? “f es un “ : “f no es un “,
       “digito hexadecimal” );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
De acuerdo con isdigit:
8 es un digito
# no es un digito

De acuerdo con isalpha:
A es una letra
b es una letra
& no es una letra
4 no es una letra

De acuerdo con isalnum:
A es un digito o una letra
8 es un digito o una letra
# no es un digito o una letra
```

**Figura 8.2** Uso de **isdigit**, **isalpha**, **isalnum** e **isxdigit**. (Parte 1 de 2.)

```
De acuerdo con isxdigit:
F es un digito hexadecimal
J no es un digito hexadecimal
7 es un digito hexadecimal
$ no es un digito hexadecimal
f es un digito hexadecimal
```

Figura 8.2 Uso de **isdigit**, **isalpha**, **isalnum** y **isxdigit**. (Parte 2 de 2.)

La figura 8.2 utiliza el operador condicional (**?:**) con cada función para determinar si una cadena **" es un "** o la cadena **" no es un "** debe imprimirse en la salida de cada carácter evaluado. Por ejemplo, la expresión

```
isdigit( '8' ) ? "8 es un " : "8 no es un "
```

indica que si **'8'** es un dígito [es decir, **isdigit** devuelve un valor verdadero (diferente de **0**)], se imprime la cadena **"8 es un"**, y si **'8'** no es un dígito (es decir, **isdigit** devuelve **0**), se imprime la cadena **"8 no es un"**.

La figura 8.3 muestra las funciones ***islower***, ***isupper***, ***tolower*** y ***toupper***. La función **islower** determina si su argumento es una letra minúscula (**a-z**). La función **isupper** determina si su argumento es una letra mayúscula (**A-Z**). La función **tolower** convierte una letra mayúscula a minúscula y devuelve la letra minúscula. Si el argumento no es una letra mayúscula, **tolower** devuelve el argumento sin cambio. La función **toupper** convierte una letra minúscula a una letra mayúscula y devuelve la letra mayúscula. Si el argumento no es una letra minúscula, **toupper** devuelve el argumento sin cambio.

```
/* Figura 8.3: fig08_03.c
   Uso de las funciones islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>

int main()
{
   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
           "De acuerdo con islower:",
           islower( 'p' ) ? "p es una " : "p no es una ",
           "letra minuscula",
           islower( 'P' ) ? "P es una " : "P no es una ",
           "letra minuscula",
           islower( '5' ) ? "5 es una " : "5 no es una ",
           "letra minuscula",
           islower( '!' ) ? "! es una " : "! no es una ",
           "letra minuscula" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
           "De acuerdo con isupper:",
           isupper( 'D' ) ? "D es una " : "D no es una ",
           "letra mayuscula",
           isupper( 'd' ) ? "d es una " : "d no es una ",
           "letra mayuscula",
           isupper( '8' ) ? "8 es una " : "8 no es una ",
           "letra mayuscula",
           isupper( '$' ) ? "$ es una " : "$ no es una ",
           "letra mayuscula" );

   printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
```

Figura 8.3 Uso de las funciones **islower**, **isupper**, **tolower** y **toupper**. (Parte 1 de 2.)

```
              "u convertida a mayuscula es ", toupper( 'u' ),
              "7 convertida a mayuscula es ", toupper( '7' ),
              "$ convertida a mayuscula es ", toupper( '$' ),
              "L convertida a minuscula es ", tolower( 'L' ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
De acuerdo con islower:
p es una letra minuscula
P no es una letra minuscula
5 no es una letra minuscula
! no es una letra minuscula

De acuerdo con isupper:
D es una letra mayuscula
d no es una letra mayuscula
8 no es una letra mayuscula
$ no es una letra mayuscula

u convertida a mayuscula es U
7 convertida a mayuscula es 7
$ convertida a mayuscula es $
L convertida a minuscula es l
```

**Figura 8.3** Uso de las funciones **islower**, **isupper**, **tolower** y **toupper**. (Parte 2 de 2.)

La figura 8.4 muestra las funciones ***isspace***, ***iscntrl***, ***ispunct***, ***isprint*** e ***isgraph***. La función **isspace** determina si su argumento es uno de los siguientes caracteres de espacio en blanco (**' '**), avance de página (**'\f'**), nueva línea (**'\n'**), retorno de carro (**'\r'**), tabulador horizontal (**'\t'**) o el tabulador vertical (**'\v'**). La función **iscntrl** determina si su argumento es uno de los siguientes *caracteres de control*; tabulador horizontal (**'\t'**), tabulador vertical (**'\v'**), avance de página (**'\f'**), alerta (**'\a'**), retroceso (**'\b'**), retorno de carro (**'\r'**) o nueva línea (**'\n'**). La función **ispunct** determina si su argumento es un *carácter de impresión* diferente del espacio, un dígito o una letra, tal como **$**, **#**, **(**, **)**, **[**, **]**, **{**, **}**, **;**, **:** o **%**. La función **isprint** determina si su argumento es un carácter que puede desplegarse en la pantalla (incluso el carácter de espacio). La función **isgraph** evalúa los mismos caracteres que **isprint**; sin embargo, no incluye el carácter espacio.

```
/* Figura 8.4: fig08_04.c
   Uso de las funciones isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>

int main()
{
   printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
       "De acuerdo con isspace:",
       "Nueva linea", isspace( '\n' ) ? " es un " : " no es un ",
       "caracter espacio en blanco", "Tabulador horizontal",
       isspace( '\t' ) ? " es un " : " no es un ",
       "caracter espacio en blanco",
       isspace( '%' ) ? "% es un " : "% no es un ",
       "caracter espacio en blanco" );
```

**Figura 8.4** Uso de las funciones **isspace**, **iscntrl**, **ispunct**, **isprint** e **isgraph**. (Parte 1 de 2.)

```
    printf( “%s\n%s%s%s\n%s%s\n\n”, “De acuerdo con iscntrl:”,
        “Nueva linea”, iscntrl( ‘\n’ ) ? “ es un “ : “ no es un “,
        “caracter de control”, iscntrl( ‘$’ ) ? “$ es un “ :
        “$ no es un “, “control character” );

    printf( “%s\n%s%s\n%s%s%s\n%s%s\n\n”,
        “De acuerdo con ispunct:”,
        ispunct( ‘;’ ) ? “; es un “ : “; no es un “,
        “caracter de puntuacion”,
        ispunct( ‘Y’ ) ? “Y es un “ : “Y no es un “,
        “caracter de puntuacion “,
        ispunct( ‘#’ ) ? “# es un “ : “# no es un “,
        “caracter de puntuacion” );

    printf( “%s\n%s%s\n%s%s%s\n\n”, “De acuerdo con isprint:”,
        isprint( ‘$’ ) ? “$ es un “ : “$ no es un “,
        “caracter de impresion”,
        “Alerta”, isprint( ‘\a’ ) ? “ es un “ : “ no es un “,
        “caracter de impresion” );

    printf( “%s\n%s%s\n%s%s%s\n”,  “De acuerdo con isgraph:”,
        isgraph( ‘Q’ ) ? “Q es un “ : “Q no es un “,
        “caracter de impresion diferente a un espacio”,
        “Espacio”, isgraph( ‘ ‘ ) ? “ es un “ : “ no es un “,
        “caracter de impresion diferente a un espacio” );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
De acuerdo con isspace:
Nueva linea es un caracter espacio en blanco
Tabulador horizontal es un caracter espacio en blanco
% no es un caracter espacio en blanco

De acuerdo con iscntrl:
Nueva linea es un caracter de control
$ no es un caracter de control

De acuerdo con ispunct:
; es un caracter de puntuacion
Y no es un caracter de puntuacion
# es un caracter de puntuacion

De acuerdo con isprint:
$ es un caracter de impresion
Alerta no es un caracter de impresion

De acuerdo con isgraph:
Q es un caracter de impresion diferente a un espacio
Espacio no es un caracter de impresion diferente a un espacio
```

**Figura 8.4** Uso de las funciones **`isspace`**, **`iscntrl`**, **`ispunct`**, **`isprint`** e **`isgraph`**. (Parte 2 de 2.)

| Prototipo de la función | Descripción de la función |
|---|---|
| `double atof( const char *ptrN );` | Convierte la cadena **ptrN** a **double**. |
| `int atoi( const char *ptrN );` | Convierte la cadena **ptrN** a **int**. |
| `long atol( const char *ptrN );` | Convierte la cadena **ptrN** a **long int**. |
| `double strtod( const char *ptrN, char **ptrFinal );` | Convierte la cadena **ptrN** a **double**. |
| `long strtol( const char *ptrN, char **ptrFinal, int base );` | Convierte la cadena **ptrN** a **long**. |
| `unsigned long strtoul( const char *ptrN, char **ptrFinal, int base );` | Convierte la cadena **ptrN** a **unsigned long**. |

**Figura 8.5** Funciones de conversión de cadenas de la biblioteca general de utilidades.

## 8.4 Funciones de conversión de cadenas

Esta sección presenta las *funciones de conversión de cadenas* de la *biblioteca general de utilidades* (**<stdlib.h>**). Estas funciones convierten cadenas de dígitos a valores enteros y de punto flotante. La figura 8.5 resume las funciones de conversión de cadenas. Observe el uso de **const** para declarar la variable **ptrN** en los encabezados de la función (que se lee de izquierda a derecha como "**ptrN** es un apuntador a una constante de carácter"); **const** especifica que el valor del argumento no podrá modificarse.

**Tip para prevenir errores 8.3**

*Cuando utilice funciones de la biblioteca general de utilidades, incluya el encabezado* **<stdlib.h>**.

La función ***atof*** (figura 8.6) convierte su argumento, una cadena que representa un número de punto flotante, a un valor **double**. La función devuelve el valor **double**. Si el valor convertido no puede representarse, por ejemplo, si el primer carácter de la cadena no es un dígito, el comportamiento de la función **atof** es indefinido.

```
/* Figura 8.6: fig08_06.c
   Uso de atof */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   double d; /* variable para almacenar la cadena convertida */

   d = atof( "99.0" );

   printf( "%s%.3f\n%s%.3f\n",
           "La cadena \"99.0\" convertida a double es ", d,
           "El valor convertido dividido entre 2 es ",
           d / 2.0 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena "99.0" convertida a double es 99.000
El valor convertido dividido entre 2 es 49.500
```

**Figura 8.6** Uso de **atof**.

```
/* Figura 8.7: fig08_07.c
   Uso de atoi */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i; /* variable para almacenar la cadena convertida */

   i = atoi( "2593" );

   printf( "%s%d\n%s%d\n",
           "La cadena \"2593\" convertida a int es ", i,
           "El valor convertido menos 593 es ", i - 593 );

   return 0; /* indica terminación exitosa  */

} /* fin de main */
```

```
La cadena "2593" convertida a int es 2593
El valor convertido menos 593 es 2000
```

**Figura 8.7** Uso de **`atoi`**.

La función ***`atoi`*** (figura 8.7) convierte su argumento, una cadena de dígitos que representa un entero, a un valor **`int`**. La función devuelve el valor **`int`**. Si el valor convertido no puede representarse, el comportamiento de la función **`atoi`** es indefinido.

La función ***`atol`*** (figura 8.8) convierte su argumento, una cadena de dígitos que representa un entero largo, a un valor **`long`**. La función devuelve el valor **`long`**. Si el valor convertido no puede representarse, el comportamiento de la función **`atol`** es indefinido. Si **`int`** y **`long`** se almacenan en cuatro bytes, las funciones **`atoi`** y **`atol`** trabajan de manera idéntica.

```
/* Figura 8.8: fig08_08.c
   Uso de atol */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   long l; /* variable para almacenar la cadena converitida */

   l = atol( "1000000" );

   printf( "%s%ld\n%s%ld\n",
           "La cadena \"1000000\" convertida a long int es ", l,
           "El valor convertido, dividido entre 2 es ", l / 2 );

  return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena "1000000" convertida a long int es 1000000
El valor convertido, dividido entre 2 es 500000
```

**Figura 8.8** Uso de **`atol`**.

```
1  /* Figura 8.9: fig08_09.c
2     Uso de strtod */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     /* inicializa el apuntador cadena */
9     const char *cadena = “51.2% son admitidos”; /* inicializa la cadena */
10
11    double d;        /* variable para almacenar la secuencia convertida */
12    char *ptrCadena;  /* crea un apuntador char */
13
14    d = strtod( cadena, &ptrCadena );
15
16    printf( “La cadena \”%s\” se convierte en \n”, cadena );
17    printf( “un valor double %.2f y la cadena \”%s\”\n”, d, ptrCadena );
18
19    return 0; /* indica terminación exitosa */
20
21 }  /* fin de main */
```

```
La cadena “51.2% son admitidos” se convierte en
un valor double 51.20 y la cadena “% son admitidos”
```

**Figura 8.9** Uso de **`strtod`**.

La función ***strtod*** (figura 8.9) convierte una secuencia de caracteres que representan un valor de punto flotante a **`double`**. La función recibe dos argumentos, una cadena (**`char *`**) y un apuntador a una cadena (**`char **`**). La cadena contiene la secuencia de caracteres que se convertirán a **`double`**. Al apuntador se le asigna la ubicación del primer carácter después de la parte convertida de la cadena. La línea 14

```
d = strtod( cadena, &ptrCadena );
```

indica que a **`d`** se le asigna el valor **`double`** convertido de la **`cadena`**, y a **`ptrCadena`** se le asigna la ubicación del primer carácter después del valor convertido (51.2) en **`cadena`**.

La función ***strtol*** (figura 8.10) convierte a **`long`** una secuencia de caracteres que representa un entero. La función recibe tres argumentos, una cadena (**`char *`**), un apuntador a una cadena y un entero. La cadena contiene la secuencia de caracteres a convertir. El apuntador se asigna a la ubicación del primer carácter después de la parte convertida de la cadena. El entero especifica la *base* del valor que se convierte. La instrucción

```
x = strtol( cadena, &ptrResiduo, 0 );
```

```
/* Figura 8.10: fig08_10.c
   Uso de strtol */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   const char *cadena = “-1234567abc”; /* inicializa el apuntador cadena */

   char *ptrResto;  /* crea un apuntador char */
   long x;          /* variable para almacenar la secuencia convertida */
```

**Figura 8.10** Uso de **`strtol`**. (Parte 1 de 2.)

```

   x = strtol( cadena, &ptrResto, 0 );

   printf( “%s\”%s\”\n%s%ld\n%s\”%s\”\n%s%ld\n”,
           “La cadena original es “, cadena,
           “El valor convertido es “, x,
           “El resto de la cadena original es “,
           ptrResto,
           “El valor convertido mas 567 es “, x + 567 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena original es “-1234567abc”
El valor convertido es -1234567
El resto de la cadena original es “abc”
El valor convertido mas 567 es -1234000
```

**Figura 8.10** Uso de **strtol**. (Parte 2 de 2.)

de la figura 8.10 indica que a **x** se le asigna el valor **long** convertido de la **cadena**. Al segundo argumento, **ptrResiduo**, se le asigna el residuo de la **cadena** después de la conversión. El uso de **NULL** para el segundo argumento provoca que se ignore el resto de la cadena. El tercer argumento, **0**, indica que el valor a convertir puede estar en formato octal (base 8), decimal (base 10) o hexadecimal (base 16). La base se puede especificar como 0 o cualquier valor entre 2 y 36. Vea el apéndice E, Sistemas de Numeración, para una explicación detallada de los sistemas de numeración octal, decimal y hexadecimal. La representación numérica de enteros desde la base 11 a la base 36 utiliza los caracteres de la A a la Z para representar los valores de 10 a 35. Por ejemplo, los valores hexadecimales pueden contener dígitos de 0 a 9 y los caracteres de la A a la F. Un entero de base 11 puede contener los dígitos de 0 a 9 y el carácter A. Un entero de base 24 puede contener los dígitos de 0 a 9 y los caracteres de A a N. Un entero de base 36 puede contener los dígitos de 0 a 9 y los caracteres de A a Z.

La función ***strtoul*** (figura 8.11) convierte a **unsigned long** una secuencia de caracteres que representa un entero de tipo **unsigned long**. La función trabaja de la misma forma que la función **strtol**. La instrucción

```
x = strtoul( cadena, &ptrResiduo, 0 );
```

de la figura 8.11 indica que a **x** se le asigna el valor **unsigned long** convertido de la **cadena**. Al segundo argumento, **&ptrResiduo**, se le asigna el resto de **cadena** después de la conversión. El tercer argumento, **0**, indica que el valor a convertirse puede estar en formato octal o hexadecimal.

```
/* Figura 8.11: fig08_11.c
   Uso de strtoul */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   const char *cadena = “1234567abc”; /* inicializa el apuntador cadena */
   unsigned long x;    /* variable to hold converted sequence */
   char *ptrResto;     /* crea un apuntador a char */

   x = strtoul( cadena, &ptrResto, 0 );
```

**Figura 8.11** Uso de **strtoul**. ( Parte 1 de 2.)

```
    printf( "%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
            "La cadena original es ", cadena,
            "El valor convertido es ", x,
            "El resto de la cadena original es  ",
            ptrResto,
            "El valor convertido menos 567 es ", x - 567 );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena original es "1234567abc"
El valor convertido es 1234567
El resto de la cadena original es  "abc"
El valor convertido menos 567 es 1234000
```

**Figura 8.11** Uso de **`strtoul`**. ( Parte 2 de 2.)

## 8.5 Funciones de entrada/salida de la biblioteca estándar

Esta sección presenta varias funciones de entrada/salida de la biblioteca estándar (**`<stdio.h>`**) especialmente para la manipulación de datos en formato carácter y en formato de cadena. La figura 8.12 resume las funciones de entrada/salida de caracteres y cadenas de la biblioteca estándar de entrada/salida.

**Tip para prevenir errores 8.4**

*Cuando utilice funciones de la biblioteca estándar de entrada/salida, incluya el encabezado* **`<stdio.h>`**.

La figura 8.13 utiliza las funciones **`gets`** y **`putchar`** para leer una línea de texto desde la entrada estándar (teclado) y para desplegar de manera recursiva los caracteres de la línea en orden inverso. La función **`gets`** lee los caracteres de la entrada en su argumento, un arreglo de tipo **`char`**, hasta que encuentra el carácter de nueva línea o de fin de archivo. Cuando termina la lectura agrega un carácter nulo (**`'\0'`**) al arreglo. La función **`putchar`** imprime el carácter que es su argumento. El programa invoca de manera recursiva a la función **`inverso`** para imprimir la línea de texto hacia atrás. Si el primer carácter del arreglo que recibe **`inverso`** es

| Prototipo de la función | Descripción de la función |
|---|---|
| `int getchar( void );` | Lee el siguiente carácter de la entrada estándar y lo devuelve como un entero. |
| `char *gets( char *s ) ;` | Lee el siguiente carácter de la entrada estándar y lo coloca en el arreglo **`s`** hasta que encuentra un carácter de nueva línea o de fin de archivo. Agrega un carácter de terminación nulo al arreglo. |
| `int putchar( int c );` | Imprime el carácter almacenado en **`c`**. |
| `int puts( const char *s ) ;` | Imprime la cadena **`s`** seguida por el carácter de nueva línea. |
| `int sprintf( char *s, const char *formato, ... );` | Equivalente a **`printf`**, excepto que la salida se almacena en el arreglo **`s`**, en lugar de imprimirse en la pantalla. |
| `int sscanf( char *s, const char *formato, ... );` | Equivalente a **`scanf`**, excepto que la entrada se lee desde el arreglo **`s`**, en lugar leerlo desde el teclado. |

**Figura 8.12** Funciones de entrada/salida de caracteres y cadenas de la biblioteca estándar.

el carácter nulo (**'\0'**), inverso regresa. De lo contrario, se llama nuevamente a **inverso** con la dirección del subarreglo que comienza en el elemento **s[1]**, y se despliega el carácter **s[0]** mediante **putchar**, cuando se completa la llamada recursiva. El orden de los dos elementos en la porción **else** de la instrucción **if** provoca que **inverso** avance hacia el carácter de terminación nulo de la cadena, antes de que se imprima un carácter. Conforme se completan las llamadas recursivas, los caracteres se despliegan en orden inverso.

```
/* Figura 8.13: fig08_13.c
   Uso de gets y putchar */
#include <stdio.h>

void inverso( const char * const ptrS ); /* prototipo */

int main()

   char enunciado[ 80 ]; /* crea un arreglo de caracteres */

   printf( "Introduzca una linea de texto:\n" );

   /* utiliza gets para leer una línea de texto */
   gets( enunciado );

   printf( "\nLa linea impresa al reves es:\n" );
   inverso( enunciado );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* imprime recursivamente los caracteres de una cadena en orden inverso */
void inverso( const char * const ptrS )
{
   /* si es el final de la cadena */
   if ( ptrS[ 0 ] == '\0' ) { /* caso base */
       return;
   } /* fin de if */
   else { /* si no es el final de la cadena */
      inverso( &ptrS[ 1 ] ); /* paso recursivo */

      putchar( ptrS[ 0 ] ); /* utiliza putchar para desplegar los caracteres */
   } /* end else */

} /* fin de la función inverso */
```

```
Introduzca una linea de texto
Caracteres y Cadenas

La linea impresa al reves es:
sanedaC y seretcaraC
```

```
Introduzca una linea de texto:
y ahi estaba yo cuando vi a Elba

La linea impresa al reves es:
ablE a iv odnauc oy abatse iha y
```

**Figura 8.13** Uso de **gets** y **putchar**.

La figura 8.14 utiliza las funciones ***getchar*** y ***puts*** para leer los caracteres desde la entrada estándar, colocados en el arreglo de caracteres **enunciado**, y para imprimir el arreglo de caracteres como una cadena. La función **getchar** lee un carácter desde la entrada estándar y devuelve el carácter como un entero. La función **puts** toma una cadena (**char ***) como argumento, e imprime la cadena seguida por un carácter de nueva línea.

```
1  /* Figura 8.14: fig08_14.c
2     Uso de getchar y puts */
3  #include <stdio.h>
4
5  int main()
6  {
7     char c;              /* variable para almacenar los caracteres
                              introducidos por el usuario */
8     char enunciado[ 80 ]; /* crea un arreglo de caracteres */
9     int i = 0;           /* inicializa el contador i */
10
11    /* indica al usuario que introduzca una línea de texto */
12    puts( "Introduzca una linea de texto:" );
13
14    /* utiliza getchar para leer cada caracter */
15    while ( ( c = getchar() ) != '\n') {
16       enunciado[ i++ ] = c;
17    } /* fin de while */
18
19    enunciado[ i ] = '\0'; /* termina la cadena */
20
21    /* utiliza puts para desplegar el enunciado */
22    puts( "\nLa linea introducida es :" );
23    puts( enunciado );
24
25    return 0; /* indica terminación exitosa */
26
27 } /* fin de main */
```

```
Introduzca una linea de texto:
Esta es una prueba

La linea introducida es :
Esta es una prueba
```

**Figura 8.14** Uso de **getchar** y **puts**.

El programa termina de introducir caracteres cuando **getchar** lee el carácter de nueva línea introducido por el usuario para finalizar la línea de texto. Agrega un carácter nulo al arreglo **enunciado** (línea 19) de manera que el arreglo puede tratarse como una cadena. Después, la función **puts** imprime la cadena contenida en **enunciado**.

La figura 8.15 utiliza la función ***sprintf*** para imprimir datos con formato en el arreglo **s** (un arreglo de caracteres). La función utiliza el mismo especificador de conversión que **printf** (vea el capítulo 9 para una explicación detallada de todas las características de impresión con formato). El programa introduce un valor **int** y un valor **double** para darles formato y para imprimirlos en el arreglo **s**. El arreglo **s** es el primer argumento de **sprintf**.

La figura 8.16 utiliza la función ***sscanf*** para leer datos con formato desde el arreglo de caracteres **s**. La función utiliza el mismo especificador de conversión que **scanf**. El programa lee un **int** y un **double** desde el arreglo **s** y almacena los valores en **x** y **y**, respectivamente. Se imprimen los valores de **x** y **y**. El arreglo **s** es el primer argumento de **sscanf**.

```
/* Figura 8.15: fig08_15.c
   Uso de sprintf */
#include <stdio.h>

int main()
{
   char s[ 80 ];  /* crea un arreglo de caracteres */
   int x;         /* valor x a introducir */
   double y;      /* valor y a introducir */

   printf( "Introduzca un entero y un double:\n" );
   scanf( "%d%lf", &x, &y );

   sprintf( s, "entero:%6d\ndouble:%8.2f", x, y );

   printf( "%s\n%s\n",
           "La salida con formato, almacenada en el arreglo s, es:", s );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca un entero y un double:
298 87.375
La salida con formato, almacenada en el arreglo s, es:
entero:   298
double:   87.38
```

**Figura 8.15** Uso de **sprintf**.

```
/* Figura 8.16: fig08_16.c
   Uso de sscanf */
#include <stdio.h>

int main()
{
   char s[] = "31298 87.375"; /* inicializa el arreglo s */
   int x;    /* valor x a introducir */
   double y; /* valor y a introducir */

   sscanf( s, "%d%lf", &x, &y );

   printf( "%s\n%s%6d\n%s%8.3f\n",
           "Los valores almacenados en el arreglo de caracteres s son:",
           "entero:", x, "double:", y );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Los valores almacenados en el arreglo de caracteres s son:
entero: 31298
double:  87.375
```

**Figura 8.16** Uso de **sscanf**.

## 8.6 Funciones de manipulación de cadenas de la biblioteca de manipulación de cadenas

La biblioteca de manipulación de cadenas (**<string.h>**) proporciona muchas funciones útiles para manipular datos de cadena (*copiar y concatenar cadenas*), *comparar cadenas, buscar* caracteres y otras cadenas *dentro de cadenas*, separar cadenas en *tokens* (separar cadenas en su piezas lógicas) y *determinar la longitud de cadenas*. Esta sección presenta las funciones de manipulación de cadenas de la biblioteca de manipulación de cadenas. Las funciones se resumen en la figura 8.17. Cada función, excepto **strncpy**, agrega el carácter nulo a su resultado.

Las funciones **strncpy** y **strncat** especifican un parámetro de tipo **size_t**, el cual es un tipo predefinido por el estándar de C como el tipo entero del valor devuelto por el operador **sizeof**.



**Tip de portabilidad 8.2**

*El tipo **size_t** es un sinónimo dependiente de la máquina para el tipo **unsigned long** o el tipo **unsigned int**.*



**Tip para prevenir errores 8.5**

*Cuando utilice funciones de la biblioteca de manipulación de cadenas, incluya el encabezado **<string.h>**.*

La función **strcpy** copia su segundo argumento (una cadena) dentro de su primer argumento (un arreglo de caracteres que debe ser lo suficientemente grande para almacenar la cadena y el carácter de terminación nulo, el cual también se copia). La función **strncpy** es equivalente a **strcpy**, excepto que **strncpy** especifica en número de caracteres a copiar desde la cadena hacia el arreglo. Observe que la función **strncpy** no necesariamente copia el carácter de terminación nulo de su segundo argumento. Un carácter de terminación nulo se escribe solamente si el número de caracteres a copiar es al menos mayor en uno que la longitud de la cadena. Por ejemplo, si **“prueba”** es el segundo argumento, se escribe un carácter de terminación nulo sólo si el tercer argumento de **strncpy** es al menos **7** (seis caracteres en **“prueba”** más el carácter de terminación nulo). Si el tercer argumento es mayor que **7**, el carácter nulo se agrega al arreglo hasta que se escriben el número total de caracteres especificados en el tercer argumento.



**Error común de programación 8.6**

*No agregar el carácter de terminación nulo al primer argumento de **strncpy**, cuando el tercer argumento es menor o igual que la longitud de la cadena del segundo argumento.*

La figura 8.18 utiliza **strcpy** para copiar la cadena completa del arreglo **x** dentro del arreglo **y**, y utiliza **strncpy** para copiar los primeros **14** caracteres del arreglo **x** dentro del arreglo **z**. Se agrega un carácter nulo (**‘\0’**) al arreglo **z**, debido a que la llamada a **strncpy** en el programa no escribe un carácter de terminación nulo (el tercer argumento es menor que la longitud de la cadena del segundo argumento).

| Prototipo de la función | Descripción de la función |
|---|---|
| **char *strcpy( char *s1, const char *s2 )** | |
| | Copia la cadena **s2** dentro del arreglo **s1**. Devuelve el valor de **s1**. |
| **char *strncpy( char *s1, const char *s2, size_t n )** | |
| | Copia al menos **n** caracteres de la cadena **s2** dentro del arreglo **s1**. Devuelve el valor de **s1**. |
| **char *strcat( char *s1, const char *s2 )** | |
| | Agrega la cadena **s2** al arreglo **s1**. El primer carácter de **s2** sobrescribe al carácter de terminación nulo de **s1**. Devuelve el valor de **s1**. |
| **char *strncat( char *s1, const char *s2, size_t n )** | |
| | Agrega al menos **n** caracteres de la cadena **s2** al arreglo **s1**. El primer carácter de **s2** sobrescribe al carácter de terminación nulo de **s1**. Devuelve el valor de **s1**. |

**Figura 8.17** Funciones de la biblioteca de manipulación de cadenas.

```
/* Figura 8.18: fig08_18.c
   Uso de strcpy y strncpy */
#include <stdio.h>
#include <string.h>

int main()
{
   char x[] = “Feliz cumpleanios a ti”; /* inicializa el arreglo de
                                           caracteres x */
   char y[ 25 ]; /* crea el arreglo de caracteres y */
   char z[ 15 ]; /* crea el arreglo de caracteres z */

   /* contenido de la copia de x en y */
   printf( “%s%s\n%s%s\n”,
      “La cadena en el arreglo x es: “, x,
      “La cadena en el arreglo y es: “, strcpy( y, x ) );

   /* copia los primeros 17 caracteres de x dentro de z. No copia el
      caracter nulo */
   strncpy( z, x, 17 );

   z[ 17 ] = ‘\0’; /* termina la cadena en z */
   printf( “La cadena en el arreglo z es: %s\n”, z );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena en el arreglo x es: Feliz cumpleanios a ti
La cadena en el arreglo y es: Feliz cumpleanios a ti
La cadena en el arreglo z es: Feliz cumpleanios
```

**Figura 8.18** Uso de **`strcpy`** y de **`strncpy`**.

La función ***`strcat`*** *agrega* su segundo argumento (una cadena) a su primer argumento (un arreglo de caracteres que contiene una cadena). El primer carácter del segundo argumento remplaza el nulo (**`‘\0’`**) que termina la cadena del primer argumento. El programador debe asegurarse de que el arreglo utilizado para almacenar la primera cadena es lo suficientemente grande para almacenar la primera cadena, la segunda cadena y el carácter de terminación nulo copiado desde la segunda cadena. La función **`strncat`** agrega un número específico de caracteres desde la segunda cadena hacia la primera cadena. Un carácter de terminación nulo se agrega automáticamente al resultado. La figura 8.19 muestra las funciones **`strcat`** y **`strncat`**.

```
/* Figura 8.19: fig08_19.c
   Uso de strcat y strncat */
#include <stdio.h>
#include <string.h>

int main()
{
   char s1[ 20 ] = “Feliz “;   /* inicializa el arreglo de caracteres s1 */
   char s2[] = “Anio Nuevo “;  /* inicializa el arreglo de caracteres s2 */
   char s3[ 40 ] = “”;         /* inicializa como vacío el arreglo de
                                  caracteres s3 */

```

**Figura 8.19** Uso de **`strcat`** y **`strncat`**. (Parte 1 de 2.)

```
    printf( "s1 = %s\ns2 = %s\n", s1, s2 );

    /* concatena s2 y s1 */
    printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );

    /* concatena los primeros 6 caracteres de s1 a s3. Coloque '\0'
       después del último caracter */
    printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );

    /* concatena s1 a s3 */
    printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
s1 = Feliz
s2 = Anio Nuevo
strcat( s1, s2 ) = Feliz Anio Nuevo
strncat( s3, s1, 6 ) = Feliz
strcat( s3, s1 ) = Feliz Feliz Anio Nuevo
```

**Figura 8.19** Uso de **`strcat`** y **`strncat`**. (Parte 2 de 2.)

## 8.7 Funciones de comparación de la biblioteca de manipulación de cadenas

Esta sección presenta las *funciones de comparación de cadenas*, **`strcmp`** y **`strncmp`**, de la biblioteca de manipulación de cadenas. La figura 8.20 contiene los prototipos de función y una breve descripción de cada función.

La figura 8.21 compara tres cadenas, utilizando las funciones **`strcmp`** y **`strncmp`**. La función **`strcmp`** compara su primer argumento de cadena con su segundo argumento de cadena, carácter por carácter. La función devuelve 0 si las cadenas son iguales, un valor negativo si la primera cadena es menor que la segunda, y un valor positivo si la primera cadena es mayor que la segunda. La función **`strncmp`** es equivalente a **`strcmp`**, con la excepción de que **`strncmp`** compara hasta el número especificado de caracteres. La función **`strncmp`** no compara los caracteres que se encuentran después del carácter nulo de una cadena. El programa imprime el valor entero devuelto por cada llamada de función.

| Prototipo de función | Descripción de la función |
|---|---|
| **`int strcmp( const char *s1, const char *s2 );`** | |
| | Compara la cadena **`s1`** con la cadena **`s2`**. La función devuelve **`0`**, menor que **`0`**, o mayor que **`0`**, si **`s1`** es igual, menor, o mayor que **`s2`**, respectivamente. |
| **`int strncmp( const char *s1, const char *s2, size_t n );`** | |
| | Compara hasta **`n`** caracteres de la cadena **`s1`** con la cadena **`s2`**. La función devuelve **`0`**, menor que **`0`**, o mayor que **`0`**, si **`s1`** es igual, menor, o mayor que **`s2`**, respectivamente. |

**Figura 8.20** Funciones de comparación de cadenas de la biblioteca de manipulación de cadenas.

```
/* Figura 8.21: fig08_21.c
   Uso de strcmp y strncmp */
#include <stdio.h>
```

**Figura 8.21** Uso de **`strcmp`** y **`strncmp`**. (Parte 1 de 2.)

```
#include <string.h>

int main()
{
   const char *s1 = "Feliz anio nuevo"; /* inicializa el apuntador a char */
   const char *s2 = "Feliz anio nuevo"; /* inicializa el apuntador a char */
   const char *s3 = "Felices fiestas";  /* inicializa el apuntador a char */

   printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
          "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
          "strcmp(s1, s2) = ", strcmp( s1, s2 ),
          "strcmp(s1, s3) = ", strcmp( s1, s3 ),
          "strcmp(s3, s1) = ", strcmp( s3, s1 ) );

   printf("%s%2d\n%s%2d\n%s%2d\n",
          "strncmp(s1, s3, 6) = ", strncmp( s1, s3, 6 ),
          "strncmp(s1, s3, 7) = ", strncmp( s1, s3, 7 ),
          "strncmp(s3, s1, 7) = ", strncmp( s3, s1, 7 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
s1 = Feliz anio nuevo
s2 = Feliz anio nuevo
s3 = Felices fiestas

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

**Figura 8.21** Uso de **strcmp** y **strncmp**. (Parte 2 de 2.)



**Error común de programación 8.7**

*Suponer que* **strcmp** *y* **strncmp** *devuelven 1 cuando sus argumentos son iguales, es un error lógico. Ambas funciones devuelven 0 (extrañamente, el equivalente del valor falso en C) para la igualdad. Por lo tanto, cuando se evalúa la igualdad de dos cadenas, el resultado de las funciones* **strcmp** *y* **strncmp** *debe compararse con 0, para determinar si las cadenas son iguales.*

Para que comprenda exactamente lo que significa que una cadena sea "más grande que" o "más pequeña que" otra cadena, considere el proceso de ordenar alfabéticamente una serie de apellidos. El lector colocaría, sin duda alguna, Martínez antes que Sánchez, debido a que, en el alfabeto, la primera letra de "Martínez" se encuentra antes que la primera letra de "Sánchez". Sin embargo, el alfabeto es más que sólo una lista de 26 letras; es una lista ordenada de caracteres. Cada letra aparece en una posición específica dentro de la lista. "Z" es más que sólo una letra del alfabeto; "Z" es específicamente la letra número 26 del alfabeto.

¿Cómo sabe la computadora que una letra en particular va antes que otra? Todos los caracteres se representan en la computadora como *códigos numéricos*; cuando la computadora compara dos cadenas, en realidad compara los códigos numéricos de los caracteres de las cadenas.



**Tip de portabilidad 8.3**

*Los códigos numéricos internos que se utilizan para representar caracteres, pueden diferir en distintas computadoras.*

En un esfuerzo por estandarizar las representaciones de caracteres, la mayoría de los fabricantes de computadoras diseñaron sus máquinas para que utilizaran uno de los dos esquemas de codificación más populares: el *ASCII* o el *EBCDIC*. ASCII significa "American Standard Code for Information Interchange", y EBCDIC significa "Extended Binary Coded Decimal Interchange Code". Existen otros esquemas de codificación, pero éstos dos son los más populares. El reciente Unicode Standard presenta una especificación para producir una codificación consistente de la gran mayoría de los caracteres y símbolos del mundo. Para aprender más sobre el Unicode Standard, visite **www.unicode.org**.

A ASCII, EBCDIC y Unicode se les denomina *conjuntos de caracteres*. La manipulación de cadenas y caracteres en realidad involucra la manipulación de los códigos numéricos adecuados, y no de los caracteres mismos. Esto explica la capacidad de intercambio de los caracteres y de pequeños enteros en C. Debido a que es importante decir que un código numérico es mayor, menor, o igual que otro, se vuelve factible relacionar varios caracteres o cadenas, uno con otro, haciendo referencia a los códigos de los caracteres. El apéndice D lista los códigos de los caracteres de ASCII.

## 8.8 Funciones de búsqueda de la biblioteca de manipulación de cadenas

Esta sección presenta las funciones de la biblioteca de manipulación de cadenas que se utilizan para buscar caracteres y cadenas en otras cadenas. La figura 8.22 resume dichas funciones. Observe que las funciones **strcspn** y **strspn** devuelven **size_t**.

| Prototipo de función | Descripción de la función |
|---|---|
| **char *strchr( const char *s, int c );** | |
| | Localiza la primera ocurrencia del carácter **c** en la cadena **s**. Si se localiza a **c**, se devuelve un apuntador a **c** en **s**. De lo contrario, se devuelve un apuntador **NULL**. |
| **size_t strcspn( const char *s1, const char *s2 );** | |
| | Determina y devuelve la longitud del segmento inicial de la cadena **s1**, que consiste en los caracteres no contenidos en la cadena **s2**. |
| **size_t strspn( const char *s1, const char *s2 );** | |
| | Determina y devuelve la longitud del segmento inicial de la cadena **s1**, que consiste sólo en los caracteres contenidos en la cadena **s2**. |
| **char *strpbrk( const char *s1, const char *s2 );** | |
| | Localiza la primera ocurrencia en la cadena **s1** de cualquier carácter de la cadena **s2**. Si se localiza un carácter de la cadena **s2**, se devuelve un apuntador al carácter de la cadena **s1**. De lo contrario, se devuelve un apuntador **NULL**. |
| **char *strrchr( const char *s, int c );** | |
| | Localiza la última ocurrencia de **c** en la cadena **s**. Si se localiza a **c**, se devuelve un apuntador a **c** en la cadena **s**. De lo contrario, se devuelve un apuntador **NULL**. |
| **char *strstr( const char *s1, const char *s2 );** | |
| | Localiza la primera ocurrencia en la cadena **s1** de la cadena **s2**. Si se localiza la cadena, se devuelve un apuntador a la cadena en **s1**. De lo contrario, se devuelve un apuntador **NULL**. |
| **char *strtok( char *s1, const char *s2 );** | |
| | Una secuencia de llamadas a **strtok** separa la cadena **s1** en "tokens" (piezas lógicas como palabras de una línea de texto) separados por caracteres contenidos en la cadena **s2**. La primera llamada contiene **s1** como el primer argumento, y las llamadas subsiguientes contienen a **NULL** como el primer argumento, para continuar separando la misma cadena. Un apuntador al token actual es devuelto por cada llamada. Si no hay más tokens cuando se llama a la función, se devuelve **NULL**. |

**Figura 8.22** Funciones de búsqueda de la biblioteca de manipulación de cadenas.

La función ***strchr*** busca la primera ocurrencia de un carácter en una cadena. Si se localiza al carácter, **strchr** devuelve un apuntador al carácter en la cadena; de lo contrario, **strchr** devuelve **NULL**. La figura 8.23 utiliza **strchr** para buscar la primera ocurrencia de **'a'** y la primera ocurrencia de **'z'** en la cadena **"Esta es una prueba"**.

```
/* Figura 8.23: fig08_23.c
   Uso de strchr */
#include <stdio.h>
#include <string.h>

int main()
{
   const char *cadena = "Esta es una prueba"; /* inicializa el apuntador
                                                 a char */
   char caracter1 = 'a'; /* inicializa el caracter1 */
   char caracter2 = 'z'; /* inicializa el caracter2 */

   /* si caracter1 se encuentra en cadena */
   if ( strchr( cadena, caracter1 ) != NULL ) {
      printf( "\'%c\' se encuentra en \"%s\".\n",
         caracter1, cadena );
   } /* fin de if */
   else { /* si no se encuentra caracter1 */
      printf( "\'%c\' no se encontro en \"%s\".\n",
         caracter1, cadena );
   } /* fin de else */

   /* si caracter2 se encuentra en cadena */
   if ( strchr( cadena, caracter2 ) != NULL ) {
   printf( "\'%c\' se encontro en \"%s\".\n",
      caracter2, cadena );
   } /* fin de if */
   else { /* si no se encontro caracter2 */
      printf( "\'%c\' no se encontro en  \"%s\".\n",
         caracter2, cadena );
   } /* fin de else */

      return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
'a' se encuentra en "Esta es una prueba".
'z' no se encontro en  "Esta es una prueba".
```

**Figura 8.23** Uso de **strchr**.

La función ***strcspn*** (figura 8.24) determina la longitud de la parte inicial de la cadena correspondiente a su primer argumento, la cual no contiene carácter alguno de la cadena de su segundo argumento. La función devuelve la longitud del segmento.

```
/* Figura 8.24: fig08_24.c
   Uso de strcspn */
#include <stdio.h>
#include <string.h>
```

**Figura 8.24** Uso de **strcspn**. (Parte 1 de 2.)

```

int main()
{
  /* inicializa dos apuntadores a char */
  const char *cadena1 = "El valor es 3.14159";
  const char *cadena2 = "1234567890";

  printf( "%s%s\n%s%s\n\n%s\n%s%u",
     "cadena1 = ", cadena1, "cadena2 = ", cadena2,
     "La longitud del segmento inicial de cadena1",
     "que no contiene caracteres de cadena2 = ",
     strcspn( cadena1, cadena2 ) );

  return 0; /* indica terminación exitosa */

/* fin de main */
```

```
cadena1 = El valor es 3.14159
cadena2 = 1234567890

La longitud del segmento inicial de cadena1
que no contiene caracteres de cadena2 = 12
```

**Figura 8.24** Uso de **strcspn**. (Parte 2 de 2.)

La función ***strpbrk*** busca en su primer argumento la primera ocurrencia de cualquiera de los caracteres de su segundo argumento. Si un carácter del segundo argumento es localizado, **strpbrk** devuelve un apuntador al carácter en el primer argumento; de lo contrario, **strpbrk** devuelve **NULL**. La figura 8.25 muestra un programa que localiza la primera ocurrencia en **cadena1** de cualquier carácter de **cadena2**.

```
/* Figura 8.25: fig08_25.c
   Uso de strpbrk */
#include <stdio.h>
#include <string.h>

int main()
{
   const char *cadena1 = "esta es una prueba"; /* inicializa el apuntador
                                                  a char */
   const char *cadena2 = "precaucion";          /* inicializa el apuntador
                                                  a char */

   printf( "%s\"%s\"\n'%c'%s\n\"%s\"\n",
      "De los caracteres en ", cadena2,
      *strpbrk( cadena1, cadena2 ),
      " aparece primero en ", cadena1 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
De los caracteres en "precaucion"
'e' aparece primero en
"esta es una prueba"
```

**Figura 8.25** Uso de **strpbrk**.

La función ***strrchr*** busca la última ocurrencia del carácter especificado en una cadena. Si se localiza al carácter, **strrchr** devuelve un apuntador al carácter en la cadena; de otro modo, **strrchr** devuelve **NULL**. La figura 8.26 muestra un programa que busca la última ocurrencia del carácter **'z'** en la cadena **"Un zoologico tiene muchos animales, incluso zorros"**.

```
/*  Figura 8.26: fig08_26.c
   Uso de strrchr */
#include <stdio.h>
#include <string.h>

int main()
{
   /* inicializa el apuntador a char */
   const char *cadena1 = "Un zoologico tiene muchos animales, incluso zorros";

   int c = 'z'; /* caracter a buscar */

   printf( "%s\n%s'%c'%s\"%s\"\n",
           "El resto de cadena1 que comienza con la",
           "ultima ocurrencia del caracter ", c,
           " es: ", strrchr( cadena1, c ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
El resto de cadena1 que comienza con la
ultima ocurrencia del caracter 'z' es: "zorros"
```

**Figura 8.26** Uso de **strrchr**.

La función ***strspn*** (figura 8.27) determina la longitud de la parte inicial de una cadena que se encuentra en su primer argumento, y que contiene sólo caracteres de la cadena en su segundo argumento. La función devuelve la longitud del segmento.

```
/* Figura 8.27: fig08_27.c
   Uso de strspn */
#include <stdio.h>
#include <string.h>

int main()
{
   /* inicializa dos apuntadores a char */
   const char *cadena1 = "El valor es 3.14159";
   const char *cadena2 = "aelv lsEro";

   printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
      "cadena1 = ", cadena1, "cadena2 = ", cadena2,
      "La longitud del segmento inicial de cadena1",
      "que contiene solamente caracteres de cadena2 = ",
      strspn( cadena1, cadena2 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 8.27** Uso de **strspn**. (Parte 1 de 2.)

```
cadena1 = El valor es 3.14159
cadena2 = aelv lsEro

La longitud del segmento inicial de cadena1
que contiene solamente caracteres de cadena2 = 12
```

**Figura 8.27** Uso de **`strspn`**. (Parte 2 de 2.)

La función ***`strstr`*** busca la primera ocurrencia de su segundo argumento de cadena en su primer argumento de cadena. Si se localiza a la segunda cadena en la primera cadena, se devuelve un apuntador a la ubicación de la cadena en el primer argumento. La figura 8.28 utiliza **`strstr`** para encontrar la cadena **`“def”`** en la cadena **`“abcdefabcdef”`**.

```
/*  Figura 8.28: fig08_28.c
    Uso de strstr */
#include <stdio.h>
#include <string.h>

int main()
{
   const char *cadena1 = “abcdefabcdef”; /* cadena de búsqueda */
   const char *cadena2 = “def”; /* cadena a buscar */

   printf( “%s%s\n%s%s\n\n%s\n%s%s\n”,
      “cadena1 = “, cadena1, “cadena2 = “, cadena2,
      “El resto de cadena1 que comienza con”,
      “la primera ocurrencia de cadena2 es: “,
      strstr( cadena1, cadena2 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
cadena1 = abcdefabcdef
cadena2 = def

El resto de cadena1 que comienza con
la primera ocurrencia de cadena2 es: defabcdef
```

**Figura 8.28** Uso de **`strstr`**.

La función ***`strtok`*** (figura 8.29) se utiliza para separar una cadena en una serie de *tokens*. Un token es una secuencia de caracteres separados por *delimitadores* (generalmente espacios o marcas de puntuación). Por ejemplo, en una línea de texto, cada palabra puede considerarse como un token, y los espacios que separan a las palabras pueden considerarse delimitadores.

```
/*  Figura 8.29: fig08_29.c
    Uso de strtok */
#include <stdio.h>
#include <string.h>

int main()
{
   /* inicializa el arreglo cadena */
```

**Figura 8.29** Uso de **`strtok`**. (Parte 1 de 2.)

```
    char cadena[] = "Este es un enunciado con 7 tokens";
    char *ptrToken; /* crea un apuntador char */

    printf( "%s\n%s\n\n%s\n",
       "La cadena a dividir en tokens es:", cadena,
       "Los tokens son:" );

    ptrToken = strtok( cadena, " " ); /* comienza la división del enunciado
                                          en tokens */

    /* continua la división en tokens, hasta que ptrToken se hace NULL */
    while ( ptrToken != NULL ) {
       printf( "%s\n", ptrToken );
       ptrToken = strtok( NULL, " " ); /* obtiene el siguiente token */
    } /* fin de while */

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena a dividir en tokens es:
Este es un enunciado con 7 tokens

Los tokens son:
Este
es
un
enunciado
con
7
tokens
```

**Figura 8.29** Uso de **strtok**. (Parte 2 de 2.)

Para separar una cadena en tokens (suponiendo que la cadena contiene más de un token), se necesitan múltiples llamadas a **strtok**. La primera llamada a **strtok** contiene dos argumentos, una cadena que va a separarse en tokens, y una cadena que contiene caracteres que separan los tokens. En la figura 8.29, la instrucción

```
ptrToken = strtok( cadena, " " ); /* comienza la división del enunciado
                                      en tokens */
```

asigna a **ptrToken** un apuntador al primer token en la **cadena**. El segundo argumento de **strtok**, **" "**, indica que los tokens de la cadena están separados por espacios. La función **strtok** busca el primer carácter de la **cadena** que no sea un carácter delimitador (un espacio). Esto comienza el primer token. La función después encuentra el siguiente carácter delimitador de la cadena y lo reemplaza por un carácter nulo (**'\0'**), para finalizar el token actual. La función **strtok** guarda un apuntador al siguiente carácter después del token de la cadena, y devuelve un apuntador al token actual.

Las llamadas subsiguientes a **strtok** continúan separando la **cadena** en tokens. Estas llamadas contienen **NULL** como su primer argumento. El argumento **NULL** indica que la llamada a **strtok** debe continuar la separación desde la ubicación en **cadena**, guardada por la última llamada a **strtok**. Si ya no hay tokens cuando se llama a **strtok**, ésta devuelve **NULL**. La figura 8.29 utiliza **strtok** para separar en tokens a la cadena **"Este es un enunciado con 7 tokens"**. Cada token se imprime de manera separada. Observe que **strtok** modifica la cadena original; por lo tanto, es necesario hacer una copia de la cadena, si es que ésta se va a usar nuevamente en el programa, después de las llamadas a **strtok**.

## 8.9 Funciones de memoria de la biblioteca de manipulación de cadenas

Las funciones de la biblioteca de manipulación de cadenas que presentamos en esta sección, manipulan, comparan y buscan bloques de memoria. Las funciones tratan a los bloques de memoria como arreglos de caracteres, y pueden manipular cualquier bloque de datos. La figura 8.30 resume las funciones de memoria de la biblioteca de manipulación de cadenas. En la explicación de funciones, un “objeto” se refiere a un bloque de datos.

| Prototipo de función | Descripción de la función |
|---|---|
| `void *memcpy( void *s1, const void *s2, size_t n );` | Copia **n** caracteres desde el objeto al que apunta **s2**, dentro del objeto al que apunta **s1**. Devuelve un apuntador al objeto resultante. |
| `void *memmove( void *s1, const void *s2, size_t n );` | Copia **n** caracteres desde el objeto al que apunta **s2** dentro del objeto al que apunta **s1**. La copia se lleva a cabo como si los caracteres primero se copiaran desde el objeto al que apunta **s2** en un arreglo temporal y después desde el arreglo temporal hacia el objeto al que apunta **s1**. Devuelve un apuntador al objeto resultante. |
| `void *memcmp( const void *s1, const void *s2, size_t n );` | Compara los primeros **n** caracteres de los objetos a los que apuntan **s1** y **s2**. La función devuelve un numero igual, menor o mayor que **0** si **s1** es igual, menor o mayor que **s2**. |
| `void *memchr( const void *s, int c, size_t n );` | Localiza la primera ocurrencia de **c** (convertida a **unsigned char**) en los primeros **n** caracteres del objeto al que apunta **s**. Si se encuentra **c**, devuelve un apuntador a **c**. De lo contrario, devuelve **NULL**. |
| `void *memset( void *s, int c, size_t n );` | Copia **c** (convertido a **unsigned char**) dentro de los primeros **n** caracteres del objeto al que apunta **s**. Devuelve un apuntador al resultado. |

**Figura 8.30** Funciones de memoria de la biblioteca de manipulación de cadenas.

Los parámetros apuntadores a estas funciones se declaran como **void***. En el capítulo 7, vimos que un apuntador a cualquier tipo de dato puede asignarse de manera directa a un apuntador de tipo **void***, y un apuntador de tipo **void*** puede asignarse de manera directa a un apuntador de cualquier tipo. Por esta razón, estas funciones pueden recibir apuntadores a cualquier tipo de dato. Debido a que un apuntador **void*** no se puede desreferenciar, cada función recibe el tamaño del argumento que especifica el número de caracteres (bytes) que procesará la función. Por sencillez, los ejemplos de esta sección manipulan arreglos de caracteres (bloques de caracteres).

La función ***memcpy*** copia un número específico de caracteres desde el objeto al que apunta su segundo argumento, dentro del objeto al que apunta el primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. El resultado de esta función es indefinido si los dos objetos se traslapan en memoria (es decir, si son parte del mismo objeto), en tales casos, utilice **memmove**. La figura 8.31 utiliza **memcpy** para copiar la cadena del arreglo **s2** al arreglo **s1**.

```
/*  Figura 8.31: fig08_31.c
    Uso de memcpy */
#include <stdio.h>
#include <string.h>

int main()
{
```

**Figura 8.31** Uso de **memcpy**. (Parte 1 de 2.)

```
   char s1[ 17 ];              /* crea el arreglo de carateres s1 */
   char s2[] = "Copia esta cadena"; /* inicializa el arreglo de caracteres s2 */

   memcpy( s1, s2, 18 );
   printf( "%s\n%s\"%s\"\n",
           "Despues de la copia de s2 en s1 con memcpy,",
           "s1 contiene ", s1 );

   return 0; /* indica terminación exitosa */

   /* fin de main */
```

```
Despues de la copia de s2 en s1 con memcpy,
s1 contiene "Copia esta cadena"
```

**Figura 8.31** Uso de **memcpy**. (Parte 2 de 2.)

La función ***memmove***, como **memcpy**, copia un número específico de bytes desde el objeto al que apunta su segundo argumento dentro del objeto al que apunta su primer argumento. La copia se lleva a cabo como si los bytes se copiaran del segundo argumento hasta un arreglo de caracteres temporal, y después se copiaran desde el arreglo temporal hasta el primer argumento. Esto permite copiar los caracteres de una parte de la cadena, dentro de otra parte de la misma cadena. La figura 8.32 utiliza **memove** para copiar los últimos **10** bytes del arreglo **x** dentro de los primeros **10** bytes del arreglo **x**.

### Error común de programación 8.8

*Las funciones de manipulación de cadenas, diferentes de **memmove**, que copian caracteres tienen un resultado indefinido cuando se lleva a cabo una copia entre partes de la misma cadena.*

La función **memcmp** (figura 8.33) compara el número específico de caracteres de su primer argumento con los caracteres correspondientes de su segundo argumento. La función devuelve un valor mayor que 0, si el primer argumento es mayor que su segundo argumento, devuelve 0 si los argumentos son iguales y devuelve un valor menor que 0, si el primer argumento es menor que el segundo argumento.

```
/* Figura 8.32: fig08_32.c
   Uso de memmove */
#include <stdio.h>
#include <string.h>

int main()
{
   char x[] = "Hogar Dulce Hogar"; /* inicializa el arreglo
                                      de caracteres x */

   printf( "%s%s\n", "La cadena en el arreglo x antes de memmove es: ", x );
   printf( "%s%s\n", "La cadena en el arreglo x despues de memmove es: ",
           memmove( x, &x[ 6 ], 11 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La cadena en el arreglo x antes de memmove es: Hogar Dulce Hogar
La cadena en el arreglo x despues de memmove es: Dulce Hogar Hogar
```

**Figura 8.32** Uso de **memmove**.

```
/* Figura 8.33: fig08_33.c
   Uso de memcmp */
#include <stdio.h>
#include <string.h>

int main()
{
   char s1[] = “ABCDEFG”; /* inicializa el arreglo de caracteres s1 */
   char s2[] = “ABCDXYZ”; /* inicializa el arreglo de caracteres s2 */

   printf( “%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n”,
          “s1 = “, s1, “s2 = “, s2,
          “memcmp( s1, s2, 4 ) = “, memcmp( s1, s2, 4 ),
          “memcmp( s1, s2, 7 ) = “, memcmp( s1, s2, 7 ),
          “memcmp( s2, s1, 7 ) = “, memcmp( s2, s1, 7 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp( s1, s2, 4 ) =  0
memcmp( s1, s2, 7 ) = -1
memcmp( s2, s1, 7 ) =  1
```

**Figura 8.33** Uso de **memcmp**.

La función ***memchr*** busca la primera ocurrencia de un byte, representado como un **unsigned char**, en el número específico de bytes de un objeto. Si encuentra el byte, la función devuelve un apuntador al byte en el objeto, de lo contrario, devuelve un apuntador **NULL**. La figura 8.34 busca el carácter (byte) **‘d’** en la cadena **“Esta es una cadena”**.

```
/* Figura 8.34: fig08_34.c
   Uso de memchr */
#include <stdio.h>
#include <string.h>

int main()
{
   const char *s = “Esta es una cadena”; /* inicializa el apuntador char */

   printf( “%s\’%c\’%s\”%s\”\n”,
          “El resto de s despues del caracter “, ‘d’,
          “ encontrado es “, memchr( s, ‘d’, 16 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
El resto de s despues del caracter ‘d’ encontrado es “dena”
```

**Figura 8.34** Uso de **memchr**.

```c
/* Figura 8.35: fig08_35.c
   Uso de memset */
#include <stdio.h>
#include <string.h>

int main()
{
   char cadena1[ 15 ] = "BBBBBBBBBBBBBB"; /* inicializa cadena1 */

   printf( "cadena1 = %s\n", cadena1 );
   printf( "cadena1 despues de memset = %s\n", memset( cadena1, 'b', 7 ) );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
cadena1 = BBBBBBBBBBBBBB
cadena1 despues de memset = bbbbbbbBBBBBBB
```

**Figura 8.35** Uso de **memset**.

La función ***memset*** copia el valor del byte en su segundo argumento, dentro del número específico de bytes del objeto al que apunta su primer argumento. La figura 8.35 utiliza **memset** para copiar **'b'** dentro de los primeros **7** bytes de **cadena1**.

## 8.10 Otras funciones de la biblioteca de manipulación de cadenas

Las dos funciones restantes de la biblioteca de manipulación de cadenas son **strerror** y **strlen**. La figura 8.36 resume las funciones **strerror** y **strlen**.

| Prototipo de función | Descripción de la función |
|---|---|
| **char *strerror( int errornum );** | Obtiene mediante **errornum** una cadena de texto del error de manera dependiente de la máquina. Devuelve un apuntador a la cadena. |
| **size_t strlen( const char *s );** | Determina la longitud de la cadena **s**. Devuelve el número de caracteres que preceden al carácter de terminación nulo. |

**Figura 8.36** Otras funciones de la biblioteca de manipulación de cadenas.

La función ***strerror*** toma un número de error y crea una cadena con el mensaje de error. Devuelve un apuntador a la cadena. La figura 8.37 muestra **strerror**.

```c
/* Figura 8.37: fig08_37.c
   Uso de strerror */
#include <stdio.h>
#include <string.h>

int main()
{
   printf( "%s\n", strerror( 2 ) );

```

**Figura 8.37** Uso de **strerror**. (Parte 1 de 2.)

```
    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
No such file or directory
```

**Figura 8.37** Uso de **`strerror`**. (Parte 2 de 2.)



### Tip de portabilidad 8.4

*El mensaje generado por* ***`strerror`*** *es dependiente de la máquina.*

La función ***`strlen`*** toma una cadena como argumento y devuelve el número de caracteres en la cadena; el carácter nulo no se incluye en la longitud. La figura 8.38 muestra la función **`strlen`**.

```
/*  Figura 8.38: fig08_38.c
    Uso de strlen */
#include <stdio.h>
#include <string.h>

int main()
{
    /* inicializa los 3 apuntadores a char */
    const char *cadena1 = "abcdefghijklmnopqrstuvwxyz";
    const char *cadena2 = "cuatro";
    const char *cadena3 = "Mexico";

    printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
       "La longitud de ", cadena1, " es ",
       ( unsigned long ) strlen( cadena1 ),
       "La longitud de ", cadena2, " es ",
       (unsigned long ) strlen( cadena2 ),
       "La longitud de ", cadena3, " es ",
       (unsigned long) strlen( cadena3 ) );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
La longitud de "abcdefghijklmnopqrstuvwxyz" es 26
La longitud de "cuatro" es 6
La longitud de "Mexico" es 6
```

**Figura 8.38** Uso de **`strlen`**.

## RESUMEN

- La función **`islower`** determina si su argumento es una letra minúscula (**`a-z`**).
- La función **`isupper`** determina si su argumento es una letra mayúscula (**`A-Z`**).
- La función **`isdigit`** determina si su argumento es un dígito (**`0-9`**).
- La función **`isalpha`** determina si su argumento es una letra mayúscula (**`A-Z`**), o una letra minúscula (**`a-z`**).
- La función **`isalnum`** determina si su argumento es una letra mayúscula (**`A-Z`**), una letra minúscula (**`a-z`**) o un dígito (**`0-9`**).
- La función **`isxdigit`** determina si su argumento es un dígito hexadecimal (**`A-F`**, **`a-f`**, **`0-9`**).
- La función **`toupper`** convierte una letra minúscula a mayúscula.

- La función **tolower** convierte una letra mayúscula a minúscula.
- La función **isspace** determina si su argumento es uno de los siguientes caracteres blancos: **' '** (espacio), **'\f'**, **'\n'**, **'\r'**, **'\t'** o **'\v'**.
- La función **iscntrl** determina si su argumento es uno de los siguientes caracteres de control: **'\t'**, **'\v'**, **'\f'**, **'\a'**, **'\b'**, **'\r'** o **'\n'**.
- La función **ispunct** determina si su argumento es un carácter de impresión diferente del espacio en blanco, un dígito o una letra.
- La función **isprint** determina si su argumento es cualquier carácter de impresión, incluso el espacio en blanco.
- La función **isgraph** determina si su argumento es cualquier carácter de impresión, diferente del espacio en blanco.
- La función **atof** convierte su argumento, una cadena con una serie de dígitos que representa un número de punto flotante, a un valor **double**.
- La función **atoi** convierte su argumento, una cadena con una serie de dígitos que representa un número entero, a un valor entero.
- La función **atol** convierte su argumento, una cadena con una serie de dígitos que representa un número entero largo, a un entero largo.
- La función **strtod** convierte una secuencia de caracteres que representan un valor en punto flotante a **double**. La función recibe dos argumentos, una cadena (**char ***) y un apuntador a **char ***. Esta cadena contiene la secuencia de caracteres a convertir, y el apuntador **char *** se asigna al resto de la cadena después de la conversión.
- La función **strtol** convierte una secuencia de caracteres que representan un entero a **long**. La función recibe tres argumentos, una cadena (**char ***), un apuntador a **char *** y un entero. La cadena contiene la secuencia de caracteres a convertir, el apuntador **char *** se asigna al resto de la cadena después de la conversión, y el entero especifica la base del valor a convertir.
- La función **strtoul** convierte una secuencia de caracteres que representan un **unsigned long**. La función recibe tres argumentos, una cadena (**char ***), un apuntador a **char *** y un entero. La cadena contiene la secuencia de caracteres a convertir, el apuntador **char *** se asigna al resto de la cadena después de la conversión, y el entero especifica la base del valor a convertir.
- La función **gets** lee caracteres desde la entrada estándar (teclado) hasta que encuentra el carácter de nueva línea o de fin de archivo. El argumento de **gets** es un arreglo de tipo **char**. Cuando termina la lectura, agrega al arreglo un carácter nulo (**'\0'**).
- La función **putchar** imprime su argumento de tipo carácter.
- La función **getchar** lee un solo carácter desde la entrada estándar y lo devuelve como un entero. Si encuentra el carácter de fin de archivo, **getchar** devuelve **EOF**.
- La función **puts** toma una cadena (**char ***) como argumento y la imprime seguida por el carácter nulo.
- La función **sprintf** utiliza los mismos especificadores de conversión que **printf**, para imprimir datos con formato dentro de un arreglo de tipo **char**.
- La función **sscanf** utiliza los mismos especificadotes de conversión que **scanf**, para leer datos con formato de una cadena de caracteres.
- La función **strcpy** copia su segundo argumento (una cadena) dentro de su primer argumento (un arreglo de caracteres). El programador debe asegurarse de que el arreglo es lo bastante grande para almacenar la cadena y su carácter de terminación nulo.
- La función **strncpy** es equivalente a **strcpy**, excepto que la llamada a **strncpy** especifica el número de caracteres que se copiarán desde la cadena hasta el arreglo de caracteres. El carácter de terminación solamente se copiará si el número de caracteres es uno más que la longitud de la cadena.
- La función **strcat** agrega su segundo argumento de cadena, incluso el carácter de terminación nulo, a su primer argumento de cadena. El primer carácter de la segunda cadena remplaza el carácter nulo (**'\0'**) de la primera cadena. El programador debe asegurarse de que el arreglo que se utiliza para almacenar la primera cadena sea lo suficientemente grande para almacenar a las dos cadenas.
- La función **strncat** agrega un número específico de caracteres desde la segunda cadena a la primera cadena. Se agrega un carácter nulo al resultado.

- La función **strcmp** compara su primer argumento de cadena con su segundo argumento de cadena, carácter por carácter. La función devuelve 0 si las cadenas son iguales, devuelve un valor negativo si la primera cadena es menor que la segunda cadena, y devuelve un valor positivo si la primera cadena es mayor que la segunda cadena.
- La función **strncmp** es equivalente a **strcmp**, excepto que **strncmp** compara un número específico de caracteres. Si el número de caracteres en una de las cadenas es menor que el número de caracteres especificados, **strncmp** compara los caracteres hasta que encuentre el carácter nulo en la cadena más corta.
- La función **strchr** busca la primera ocurrencia de un carácter dentro de una cadena. Si se encuentra el carácter, **strchr** devuelve un apuntador al carácter en la cadena; de lo contrario, **strchr** devuelve **NULL**.
- La función **strcspn** determina la longitud de la parte inicial de la cadena de su primer argumento, que no contenga carácter alguno de la segunda cadena del segundo argumento. La función devuelve la longitud del segmento.
- La función **strpbrk** busca la primera ocurrencia en el primer argumento de cualquier carácter en su segundo argumento. Si encuentra un carácter de su segundo argumento, **strpbrk** devuelve un apuntador al carácter; de lo contrario, **strpbrk** devuelve **NULL**.
- La función **strrchr** busca la última ocurrencia de un carácter en la cadena. Si encuentra el carácter, **strrchr** devuelve un apuntador al carácter en la cadena; de lo contrario, **strrchr** devuelve **NULL**.
- La función **strspn** determina la longitud de la parte inicial de la cadena de su primer argumento, que contenga sólo caracteres de la cadena de su segundo argumento. La función devuelve la longitud del segmento.
- La función **strstr** busca la primera ocurrencia de su segundo argumento de cadena dentro de su primer argumento de cadena. Si encuentra la segunda cadena dentro de la primera, devuelve un apuntador a la ubicación de la cadena del primer argumento.
- Una secuencia de llamadas a **strtok** rompe la cadena **s1** en tokens (elementos) separados por caracteres contenidos en la cadena **s2**. La primera llamada contiene a **s1** como primer argumento, y las llamadas subsiguientes continúan la división de la misma cadena con **NULL** como primer argumento. Cada llamada devuelve un apuntador al token actual. Si no existen tokens cuando se invoca a la función, la función devuelve un apuntador a **NULL**.
- La función **memcpy** copia un número específico de caracteres desde el objeto al cual apunta el segundo argumento hacia el objeto al cual apunta el primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. Los apuntadores se reciben desde **memcpy** como apuntadores **void** y se convierten a apuntadores **char** para que se puedan utilizar en la función. La función **memcpy** manipula los bytes del objeto como caracteres.
- La función **memmove** copia un número específico de bytes desde el objeto al cual apunta el segundo argumento hacia el objeto al cual apunta el primer argumento. La copia se lleva a cabo como si los dos bytes se copiaran desde el segundo argumento hacia un arreglo de caracteres temporal, y después se copiaran desde un arreglo temporal hacia el primer argumento.
- La función **memcmp** compara el número especificado de caracteres de su primer y segundo argumento.
- La función **memchr** busca la primera ocurrencia de un byte, representado como un **unsigned char**, en el número especificado de bytes de un objeto. Si encuentra el byte, devuelve un apuntador hacia dicho byte; de lo contrario, devuelve un apuntador **NULL**.
- La función **memset** copia su segundo argumento, tratado como un **unsigned char**, hacia un número específico de bytes al que apunta su primer argumento.
- La función **strerror** obtiene mediante **errornum** una cadena de texto del error de manera dependiente de la máquina. Devuelve un apuntador a la cadena.
- La función **strlen** toma una cadena como argumento y devuelve el número de caracteres en la cadena; en la longitud de la cadena no se incluye el carácter de terminación nulo.

## *TERMINOLOGÍA*

agregar cadenas a otras cadenas
ASCII
**atof**
**atoi**
**atol**
biblioteca de manipulación de cadenas
biblioteca general de utilerías utilidades generales
cadena
cadena de búsqueda
carácter de control
carácter imprimible de impresión
caracteres de espacios en blanco
código de carácter
código numérico para la representación numérica de un carácter
comparación de cadenas
concatenación de cadenas
conjunto de caracteres

constante de cadena
constante de carácter
copia de cadenas
**ctype.h**
delimitador
dígitos hexadecimales
división separación de cadenas en tokens (tokenización)
**EOF**
funciones de búsqueda
funciones de comparación de cadenas
funciones de conversión de cadenas
funciones de manipulación de cadenas
**getchar**
**gets**
**isalnum**
**isalpha**
**iscnctrl**
**isdigit**
**isgraph**
**islower**
**isprint**
**ispunct**
**isspacec**
**isupper**
**isxdigit**
literal
literal de cadena
longitud de de una cadena
**memcmp**
**memcpy**
**memchr**
**memmove**
**memset**
procesamiento de cadenas
procesamiento de palabras
**putchar**
**puts**
**sprintf**
**sscanf**
**stdio.h**
**stdlib.h**
**strcat**
**strcmp**
**strcpy**
**strcspn**
**strchr**
**strerror**
**string.h**
**strlen**
**strncat**
**strncpm**
**strncpy**
**strpbrk**
**strrchr**
**strspn**
**strstr**
**strtod**
**strtok**
**strtol**
**strtoul**
**tolower**
**toupper**
**Unicodte**

## *ERRORES COMUNES DE PROGRAMACIÓN*

**8.1** No almacenar suficiente espacio en un arreglo de caracteres para almacenar el carácter nulo que termina una cadena, es un error.

**8.2** Imprimir una "cadena" que no contiene el carácter de terminación nulo es un error.

**8.3** Procesar un solo carácter como una cadena. Una cadena es un apuntador, probablemente un entero de tamaño respetable. Sin embargo, un carácter es un entero pequeño (en el rango de valores ASCII 0-255). En muchos sistemas esto provoca un error, debido a que las direcciones de memoria baja se reservan para propósitos especiales tales como los manipuladores de interrupciones del sistema operativo, por lo que ocurren "violaciones de acceso".

**8.4** Pasar un carácter como argumento a una función cuando se espera una cadena, es un error de sintaxis.

**8.5** Pasar una cadena como un argumento a una función cuando se espera un carácter, es un error de sintaxis.

**8.6** No agregar el carácter de terminación nulo al primer argumento de **strncpy**, cuando el tercer argumento es menor o igual que la longitud de la cadena en el segundo argumento.

**8.7** Suponer que **strcmp** y **strncmp** devuelven 1 cuando sus argumentos son iguales, es un error lógico. Ambas funciones devuelven 0 (extrañamente, el equivalente del valor falso en C) para la igualdad. Por lo tanto, cuando se evalúa la igualdad de dos cadenas, el resultado de las funciones **strcmp** y **strncmp** debe compararse con 0, para determinar si las cadenas son iguales.

**8.8** Las funciones de manipulación de cadenas, diferentes de **memmove**, que copian caracteres tienen un resultado indefinido cuando se lleva a cabo una copia entre partes de la misma cadena.

## *TIPS PARA PREVENIR ERRORES*

**8.1** Cuando almacene una cadena de caracteres dentro de un arreglo, asegúrese de que el arreglo sea lo suficientemente grande para almacenar la cadena más larga que se vaya a guardar. C permite almacenar cadenas de cualquier longitud. Si una cadena es más grande que el arreglo de caracteres en el cual se va a almacenar, los caracteres más allá del final del arreglo sobrescribirán los datos siguientes en la memoria al arreglo.

**8.2** Cuando utilice funciones de la biblioteca de manipulación de caracteres, incluya el encabezado **<ctype.h>**.

**8.3** Cuando utilice funciones de la biblioteca general de utilidades, incluya el encabezado **<stdlib.h>**.

**8.4** Cuando utilice funciones de la biblioteca estándar de entrada/salida, incluya el encabezado **<stdio.h>**.

**8.5** Cuando utilice funciones de la biblioteca de manipulación de cadenas, incluya el encabezado **<string.h>.**

## TIPS DE PORTABILIDAD

**8.1** Cuando se inicializa una variable de tipo **char*** con una literal de cadena, es posible que algunos compiladores coloquen la cadena en un lugar de la memoria, en donde ésta no se pueda modificar. Si necesitara modificar una literal de cadena, podría almacenarla en un arreglo de caracteres para garantizar que pueda modificarla en cualquier sistema.

**8.2** El tipo **size_t** es un sinónimo dependiente de la máquina para el tipo **unsigned long** o el tipo **unsigned int**.

**8.3** Los códigos numéricos internos que se utilizan para representar caracteres, pueden diferir en distintas computadoras.

**8.4** El mensaje generado por **strerror** es dependiente de la máquina.

## EJERCICIOS DE AUTOEVALUACIÓN

**8.1** Escriba una instrucción sencilla para llevar a cabo cada una de las siguientes tareas. Suponga que las variables **c**, **x**, **y** y **z** (las cuales almacenan un carácter) son de tipo **int**, que las variables **d**, **e** y **f** son de tipo **double**, que la variable **ptr** es de tipo **char *** y que los arreglos **s1[ 100 ]** y **s2[ 100 ]** son de tipo **char**.

a) Convierta el carácter almacenado en **c** a mayúscula. Asigne el resultado a la variable **c**.
b) Determine si el valor de la variable **c** es un dígito. Utilice el operador condicional como lo muestran las figuras 8.2 a 8.4 para imprimir **"es un "** o **"no es un "**, cuando despliegue el resultado.
c) Convierta la cadena **"1234567"** a **long**, e imprima el valor.
d) Determine si el valor de la variable **c** es un carácter de control. Utilice el operador condicional para imprimir **"es un "** o **"no es un "**, cuando despliegue el resultado.
e) Lea una línea de texto del arreglo **s1** desde el teclado. No utilice **scanf**.
f) Imprima la línea de texto almacenada en el arreglo **s1**. No utilice **printf**.
g) Asigne a **ptr** la ubicación de la última ocurrencia de **c** en **s1**.
h) Imprima el valor de la variable **c**. No utilice **printf**.
i) Convierta la cadena **"8.63582"** a **double**, e imprima el valor.
j) Determine si el valor de **c** es una letra. Utilice el operador condicional para imprimir **" es un "** o **" no es un "**, cuando despliegue el resultado.
k) Lea un carácter desde el teclado y almacénelo en la variable de carácter **c**.
l) Asigne a **ptr** la ubicación de la primera ocurrencia de **s2** en **s1**.
m) Determine si el valor de la variable **c** es un carácter de impresión. Utilice el operador condicional para imprimir **" es un "** o **" no es un "** cuando despliegue el resultado.
n) Lea tres valores **double** dentro de las variables **d**, **e** y **f** de la cadena **"1.27 10.3 9.432"**.
o) Copie la cadena almacenada en el arreglo **s2** hacia el arreglo **s1**.
p) Asigne **ptr** a la ubicación de la primera ocurrencia en **s1** de cualquier carácter de **s2**.
q) Compare la cadena en **s1** con la cadena en **s2**. Imprima el resultado.
r) Asigne a **ptr** la ubicación de la primera ocurrencia de **c** en **s1**.
s) Utilice **sprintf** para imprimir los valores de las variables enteras **x**, **y** y **z** dentro del arreglo **s1**. Cada valor debe imprimirse con un ancho de campo de **7** posiciones.
t) Agregue 10 caracteres de la cadena **s2** a la cadena **s1**.
u) Determine la longitud de la cadena en **s1**. Imprima el resultado.
v) Convierta la cadena **"-21"** a **int**, e imprima el valor.
w) Asigne **ptr** a la ubicación del primer elemento (token) en **s2**. Los tokens de la cadena **s2** se separan con comas (**,**).

**8.2** Muestre dos métodos diferentes para inicializar el arreglo de caracteres vocales con la cadena de vocales **"AEIOU"**.

**8.3** Al ejecutarse, ¿qué imprime cada una de las siguientes instrucciones en C? Si la instrucción contiene un error, descríbalo e indique cómo corregirlo. Suponga las siguientes definiciones de variables:

```
char s1[ 50 ] = "juan", s2[ 50 ] = " lola", s3[ 50 ], *ptrS;
```

```
a) printf( "%c%s", toupper( s1[ 0 ] ), &s1[ 1 ] );
b) printf( "%s", strcpy( s3, s2 ) );
c) printf( "%s",
       strcat( strcat( strcpy( s3, s1 ), " y " ), s2 ) );
d) printf( "%u", strlen( s1 ) + strlen( s2 ) );
e) printf( "%u", strlen( s3 ) );
```

**8.4** Encuentre el error en cada uno de los siguientes segmentos de programa y explique cómo corregirlos.

a)
```
char s[ 10 ];
strncpy( s, "hola", 4 );
printf( "%s\n", s );
```
b)
```
printf( "%s", 'a' );
```
c)
```
char s[ 17 ];
strcpy( s, "bienvenido a casa" );
```
d)
```
if ( strcmp( cadena1, cadena2 ) )
printf( "Las cadenas son iguales \n" );
```

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**8.1**

a) `c = toupper( c );`

b)
```
printf( "'%c'%sdigito\n",
        c, isdigit( c ) ? " es un " : " no es un " );
```
c) `printf( "%ld\n", atol( "1234567" ) );`

d)
```
printf( "'%c'%scaracter de control\n",
        c, iscntrl( c ) ? " es un " : " no es un " );
```
e) `gets ( s1 );`

f) `puts ( s1 );`

g) `ptr = strrchr( s1, c );`

h) `putchar( c );`

i) `printf( "%f\n", atof( "8.63582" ) );`

j)
```
printf( "'%c'%sletra\n",
c, isalpha( c ) ? " es una " : " no es una " );
```
k) `c = getchar();`

l) `ptr = strstr( s1, s2 );`

m)
```
printf( "'%c'%scaracter de impresion\n",
      c, isprint ( c ) ? " es un " : " no es un " );
```
n) `sscanf( "1.27 10.3 9.432", "%f%f%f", &d, &e, &f );`

o) `strcpy( s1, s2);`

p) `ptr = strpbrk( s1, s2 );`

q) `printf( "strcmp( s1, s2 ) = %d\n", strcmp( s1, s2 ) );`

r) `ptr = strchr( s1, c );`

s) `sprintf( s1, "%7d%7d 7d", x, y, z );`

t) `strncat( s1, s2, 10 );`

u) `printf( "strlen(s1) = %u\n", strlen( s1 ) );`

v) `printf( "%d\n", atoi( "-21" ) );`

w) `ptr = strtok( s2, ",");`

**8.2**
```
char vocales[] = "AEIOU";
char vocales[] = { 'A', 'E', 'I', 'O', 'U' };
```

**8.3**

a) **juan**

b) **lola**

c) **juan y lola**

d) **8**

e) **11**

**8.4**

a) Error: la función **strncpy** no escribe el carácter de terminación nulo para el arreglo **s**, debido a que el tercer argumento es igual a la longitud de la cadena **"hola"**.
Corrección: haga el tercer argumento de **strncpy** igual a **5**, o asigne el carácter nulo **'\0'** a **s[ 5 ]**.

b) Error: intentar imprimir una constante de carácter como una cadena.
Corrección: utilice **%c** para desplegar el carácter, o remplace **'a'** con **"a"**.

c) Error: el arreglo de caracteres **s** no es lo suficientemente largo para almacenar el carácter de terminación nulo.
Corrección: declare el arreglo con más elementos.

d) Error: la función **strcmp** devuelve **0** si las cadenas son iguales; por lo tanto, la condición en la instrucción **if** es falsa y no se ejecutará la instrucción **printf**.
Corrección: en la condición, compare el resultado de **strcmp** con **0**.

## *EJERCICIOS*

**8.5** Escriba un programa que lea un carácter desde el teclado y que pruebe el carácter con cada una de las funciones de la biblioteca de manipulación de caracteres. El programa debe imprimir el valor devuelto por cada función.

**8.6** Escriba un programa que lea una línea de texto mediante la función **`gets`** y que la introduzca en el arreglo **`s[ 100 ]`**. Muestre la línea de texto con letras mayúsculas y con letras minúsculas.

**8.7** Escriba un programa que lea cuatro cadenas que representen enteros, que convierta las cadenas a enteros, que sume los valores, y que imprima el total de los cuatro valores.

**8.8** Escriba un programa que lea cuatro cadenas que representen valores en punto flotante, que convierta las cadenas a valores **`double`**, que sume los valores y que imprima el total de los cuatro valores.

**8.9** Escriba un programa que utilice la función **`strcmp`** para comparar dos cadenas introducidas por el usuario. El programa debe establecer si la primera cadena es menor, igual o mayor que la segunda cadena.

**8.10** Escriba un programa que utilice la función **`strncmp`** para comparar dos cadenas introducidas por el usuario. El programa debe introducir el número de caracteres a comparar. El programa debe establecer si la primera cadena es menor, igual o mayor que la segunda cadena.

**8.11** Escriba un programa que utilice la generación de números aleatorios para crear oraciones. El programa debe utilizar cuatro arreglos de apuntadores a **`char`** llamados, **`articulo`**, **`sustantivo`**, **`verbo`** y **`preposicion`**. El programa debe crear una oración mediante la selección de una palabra al azar de cada arreglo en el siguiente orden: **`articulo`**, **`sustantivo`**, **`verbo`**, **`preposicion`**, **`articulo`** y **`sustantivo`**. Al elegir cada palabra, ésta se debe concatenar a las palabras previas en un arreglo lo suficientemente grande para almacenar una oración completa. Las palabras deben separarse con espacios. Cuando se imprime la oración final, ésta debe comenzar con una letra mayúscula y terminar con punto. El programa debe generar 20 oraciones.

Los arreglos deben rellenarse de la siguiente manera: El arreglo **`articulo`** debe contener los artíclos **`“el”`** **`“la”`**, **`“un”`**, **`“algun”`** y **`“cualquiera”`**; el arreglo **`sustantivo`** debe contener los sustantivos **`“nino”`**, **`“nina”`**, **`“perro”`**, **`“pueblo”`** y **`“carro”`**; el arreglo **`verbo`** debe contener los verbos **`“condujo”`**, **`“brinco”`**, **`“corrio”`**, **`”camino”`**, y **`“salto”`**; el arreglo **`preposición`** debe contener la preposiciones **`“hacia”`**, **`”desde”`**, **`“sobre”`**, **`“bajo”`** y **`“en”`**.

Cuando escriba su programa y ya funcione, modifíquelo para producir una historia corta que consista en varias de estas oraciones. (¿Qué tal la posibilidad de un escritor de términos aleatorios?)

**8.12** (*Rimas*.) Una rima es un verso humorístico de 5 líneas en el cual, la primera y la segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Mediante el uso de técnicas similares a las desarrolladas en el ejercicio 8.11, escriba un programa que genere rimas al azar. ¡Depurar el programa para generar buenas rimas es un problema desafiante, pero el resultado valdrá la pena!

**8.13** Escriba un programa que codifique frases en español al latín cerdo. El latín cerdo es una forma de codificación del lenguaje que con frecuencia se utiliza para el entretenimiento. Existen muchas variantes del método utilizado para formar frases en latín cerdo. Por sencillez, utilice el siguiente algoritmo:

Para formar una frase en latín cerdo, a partir de una frase del español, divida la frase en tokens (palabras) mediante la función **`strtok`**. Para traducir cada palabra en español a latín cerdo, coloque la primera letra de la palabra en español al final de la misma palabra, y agregue las letras **`“ay”`**. Así, la palabra **`“salta”`** se convierte en **`“altasay”`**, la palabra **`“el”`** se convierte en **`“leay”`** y la palabra **`“computadora”`** se convierte en **`“omputadoracay”`**. Los espacios entre las palabras permanecen. Suponga lo siguiente: la frase en español consiste en palabras separadas por espacios en blanco, no existen signos de puntuación, y todas las palabras tienen dos o más letras. La función **`imprimePalabraLatin`** debe desplegar cada palabra. [*Pista:* Cada vez que se encuentre un token en la llamada a **`strtok`**, pase el apuntador del token a la función **`imprimePalabraLatin`**, e imprima la palabra en latín cerdo.]

**8.14** Escriba un programa que introduzca un número telefónico como una cadena de la forma **`(555) 555-5555`**. El programa debe utilizar la función **`strtok`** para extraer el código de área como un token, los primeros tres dígitos del número telefónico como un token y también los últimos cuatro dígitos del número telefónico. Los siete dígitos del número se deben concatenar en una sola cadena. El programa debe convertir la cadena del código de área a **`int`**, y convertir la cadena del número telefónico en un **`long`**. Tanto el código del área como el número telefónico deben imprimirse.

**8.15** Escriba un programa que introduzca una línea de texto, que divida en tokens la línea por medio de la función **`strtok`** y que muestre los tokens en orden inverso.

**8.16** Escriba un programa que introduzca una línea de texto y una cadena de búsqueda desde el teclado. Mediante el uso de la función **`strstr`**, localice la primera ocurrencia de la cadena de búsqueda en la línea de texto, y asigne la ubicación a la variable **`ptrBusca`** de tipo **`char *`**. Si encuentra la cadena de búsqueda, imprima el resto de la línea de texto, comenzando con la cadena de búsqueda. Luego, utilice de nuevo **`strstr`** para localizar la siguiente ocurrencia de la cadena de búsqueda en la línea de texto. Si existe una segunda ocurrencia, imprima el resto de la línea de texto, comenzando con la segunda ocurrencia. [*Pista:* La segunda llamada a **`strstr`** debe contener **`ptrBusca + 1`** como su primer argumento.]

**8.17** Escriba un programa basado en el ejercicio 8.16 que introduzca varias líneas de texto y que busque una cadena; utilice la función **`strstr`** para determinar el número total de ocurrencias de la cadena en las líneas de texto. Imprima el resultado.

**8.18** Escriba un programa que introduzca varias líneas de texto y busque un carácter, y utilice la función **`strchr`** para determinar el total de ocurrencias del carácter en las líneas de texto.

**8.19** Escriba un programa basado en el programa del ejercicio 8.18 que introduzca varias líneas de texto y que utilice la función **`strchr`** para determinar el total de ocurrencias de cada letra del alfabeto en las líneas de texto. Las letras mayúsculas y minúsculas deben contarse juntas. Almacene el total de cada letra dentro de un arreglo e imprima los valores de forma tabular, una vez determinados dichos totales.

**8.20** Escriba un programa que introduzca varias líneas de texto y que utilice **`strtok`** para contar el número total de palabras. Asuma que las palabras se separan por espacios o por caracteres de nueva línea.

**8.21** Utilice las funciones de comparación de cadenas que explicamos en la sección 8.6 y las técnicas de ordenamiento de arreglos desarrolladas en el capítulo 6 para escribir un programa que ordene alfabéticamente una lista de cadenas. Utilice los nombres de 10 o 15 ciudades de su región como datos de su programa.

**8.22** La tabla del apéndice D muestra las representaciones de los códigos numéricos correspondientes a los caracteres en el conjunto de caracteres ASCII. Estudie esta tabla y establezca si cada una de las siguientes frases es *verdadera* o *falsa*.

a) La letra **`“A”`** se encuentra antes de la letra **`“B”`**.
b) El dígito **`“9”`** se encuentra antes del dígito **`“0”`**.
c) Los símbolos comunes para la suma, resta, multiplicación y división se encuentran antes de cualquier dígito.
d) Los dígitos se encuentran antes que las letras.
e) Si un programa de clasificación ordena las cadenas en secuencia ascendente, entonces el programa colocará el símbolo del paréntesis derecho antes que el símbolo del paréntesis izquierdo.

**8.23** Escriba un programa que lea una serie de cadenas y que imprima solamente aquellas cadenas que comiencen con la letra **`“b”`**.

**8.24** Escriba un programa que lea una serie de cadenas y que imprima solamente aquellas cadenas que terminen con las letras **`“ed”`**.

**8.25** Escriba un programa que introduzca un código ASCII y que imprima su carácter correspondiente. Modifique este programa de manera que genere todas las posibilidades para códigos de tres dígitos en el rango de 000 a 255, e intente imprimir su carácter correspondiente. ¿Qué sucede cuando ejecutamos este programa?

**8.26** Utilice como guía la tabla del conjunto de caracteres ASCII del apéndice D, y escriba sus propias versiones de las funciones para la manipulación de cadenas de la figura 8.1.

**8.27** Escriba sus propias versiones de las funciones de la figura 8.5 para convertir caracteres a números.

**8.28** Escriba dos versiones para cada una de las funciones para copiar cadenas de la figura 8.17. La primera versión debe utilizar subíndices de arreglos, y la segunda versión debe utilizar apuntadores y aritmética de apuntadores.

**8.29** Escriba sus propias versiones de las funciones **`getchar`**, **`gets`**, **`putchar`** y **`puts`** descritas en la figura 8.12.

**8.30** Escriba dos versiones de cada función de comparación de cadenas de la figura 8.20. La primera versión debe utilizar arreglos y subíndices, y la segunda versión debe utilizar apuntadores y aritmética de apuntadores.

**8.31** Escriba sus propias versiones de las funciones de la figura 8.22 para búsqueda de cadenas.

**8.32** Escriba sus propias versiones de las funciones de la figura 8.30 para manipulación de bloques de memoria.

**8.33** Escriba dos versiones de la función **`strlen`** de la figura 8.36. La primera versión debe utilizar arreglos y subíndices, y la segunda versión debe utilizar apuntadores y aritmética de apuntadores.

## *SECCION ESPECIAL: EJERCICIOS AVANZADOS DE MANIPULACIÓN DE CADENAS*

Los ejercicios anteriores son clave para el libro y están diseñados para evaluar su comprensión sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de problemas avanzados e

intermedios. Usted encontrará estos ejercicios desafiantes pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una o dos horas de programación e implementación. Otros son útiles para trabajos de laboratorio que requieren dos o tres semanas de estudio e implementación. Algunos son proyectos finales desafiantes.

**8.34** (*Análisis de texto*.) La disponibilidad de computadoras con capacidades para manipular cadenas ha originado algunos métodos para analizar los escritos de grandes autores. Se ha puesto mucha atención en el hecho de si William Shakespeare en realidad vivió. Algunos estudiosos creen que existe suficiente evidencia que indica que en realidad Christopher Marlowe escribió los escritos adjudicados a Shakespeare. Los investigadores aplican tres métodos para analizar los textos mediante una computadora.

a) Escriba un programa que lea varias líneas de texto y que imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

```
Ser, o no ser: he ahí el dilema
```

contiene dos “a”, ninguna “b”, ninguna “c”, una “d”, etcétera.

b) Escriba un programa que lea varias líneas de texto y que imprima una tabla que indique el número de palabras de una sola letra, de dos letras, de tres letras,…, que aparecen en el texto. Por ejemplo, la frase

```
¿Qué es más noble para el espíritu?
```

contiene

| Longitud de la palabra | Ocurrencias |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 2 |
| 4 | 1 |
| 5 | 1 |
| 6 | 0 |
| 7 | 0 |
| 8 | 1 |

c) Escriba un programa que lea varias líneas de texto y que imprima una tabla que indique el número de ocurrencias de cada palabra diferente en el texto. La primera versión de su programa debe incluir las palabras de la tabla en el mismo orden en el que aparecen en el texto. Intente una impresión más interesante (y útil) en la que las palabras se ordenen de manera alfabética. Por ejemplo, las líneas:

```
Ser, o no ser: he ahí el dilema
¿Qué es más noble para el espíritu?
```

contiene dos veces la palabra ser, dos veces la palabra **“el”**, una vez la palabra **“dilema”**, etcétera.

**8.35** (*Procesamiento de palabras*.) El tratamiento tan detallado sobre la manipulación de cadenas en el libro obedece, en gran medida, al crecimiento del procesamiento de palabras en los años recientes. Una importante función en el procesamiento de palabras es la *justificación*; la alineación de palabras a los márgenes derecho e izquierdo de una página. Esto genera una vista profesional del documento y da la apariencia de haber sido impresa en imprenta y no en una máquina de escribir. La justificación se puede llevar a cabo en la computadora mediante la inserción de uno o más espacios en blanco entre cada una de las palabras en la línea, de modo que la palabra más a la derecha se alinee con el margen derecho.

Escriba un programa que lea varias líneas de texto y que imprima el texto en formato justificado. Suponga que el texto se imprime en una hoja de papel de 8 1/2 pulgadas de ancho y con márgenes de una pulgada, tanto a la derecha como a la izquierda de la hoja. Suponga que la computadora imprime 10 caracteres por pulgada horizontal. Por tal motivo, su programa debe imprimir 6 1/2 pulgadas de texto o 65 caracteres por línea.

**8.36** (*Impresión de fechas en varios formatos*.) Por lo general, las fechas se imprimen en diferentes formatos en la correspondencia de negocios. Los dos formatos más comunes son:

```
21/07/2003 y 21 de julio del 2003
```

Escriba un programa que lea la fecha en el primer formato y que la imprima en el segundo formato.

**8.37** (*Protección de Cheques*.) Con frecuencia se utilizan las computadoras como sistemas de verificación de cheques, tales como aplicaciones de nóminas o cuentas por pagar. Muchas historias extrañas circulan en torno a la impresión errónea de cheques por montos que exceden a un millón de dólares. Muchos sistemas de impresión de cheques imprimen dichos montos extraños debido a errores humanos o errores de la máquina. Por supuesto, los diseñadores de sistemas hacen muchos esfuerzos para construir controles dentro de sus sistemas para prevenir la emisión de cheques erróneos.

Otro problema serio es la alteración intencional del monto de un cheque por parte de alguien que pretende cobrar dicho cheque de manera fraudulenta. Para prevenir que el monto sea alterado, la mayoría de los sistemas computarizados de impresión de cheques emplean una técnica llamada *protección de cheques*.

Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir el monto. Suponga que un cheque contiene nueve espacios en blanco en los que se supone que la computadora imprime el monto de un pago semanal. Si el monto es grande, entonces los nueve espacios serán ocupados, por ejemplo:

```
11,230.60   (monto del cheque)
---------
123456789   (números de posición)
```

Por otro lado, si el monto es menor que $1000, entonces quedarán varios espacios en blanco. Por ejemplo:

```
    99.87
---------
123456789
```

contiene tres espacios en blanco. Si el cheque se imprime con espacios en blanco, es más fácil que alguien altere el monto del cheque. Para prevenir que un cheque sea alterado, muchos sistemas de impresión de cheques insertan *asteriscos* al principio para proteger el monto de la siguiente manera:

```
****99.87
---------
123456789
```

Escriba un programa que introduzca el monto a imprimir en el cheque y después imprima, si es necesario, el monto en formato protegido con asteriscos al principio. Suponga un total de nueve espacios disponibles para la impresión del monto.

**8.38** (*Impresión del equivalente en palabras del monto del cheque*.) Para continuar con el tema del ejemplo anterior, reiteramos la importancia de diseñar sistemas de impresión de cheques que prevengan la alteración de sus montos. Un método común de seguridad requiere que el monto del cheque se escriba en números y "deletreado" en palabras. Incluso si alguien es capaz de alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en palabras.

Muchos sistemas de cómputo para impresión de cheques no imprimen el monto del cheque en palabras. Quizá la principal razón para esta omisión sea el hecho de que la mayoría de los lenguajes de alto nivel utilizados en aplicaciones comerciales no contienen las características adecuadas de manipulación de cadenas. Otra razón es la lógica involucrada en la escritura de los equivalentes en palabras de los montos de los cheques.

Escriba un programa que introduzca un monto numérico de cheque y que escriba el equivalente en palabras de dicho monto. Por ejemplo, el monto 112.34 se debe escribir como

```
CIENTO DOCE y 34/100
```

**8.39** (*Clave Morse*.) Tal vez el esquema de código más famoso del mundo sea la clave Morse, desarrollado por Samuel Morse en 1832 para uso del sistema telegráfico. La clave Morse asigna una serie de puntos y guiones a cada letra del alfabeto, a cada dígito, y a algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas basados en sonido, el punto representa un sonido corto y el guión representa un sonido largo. En los sistemas basados en luz y en los sistemas basados en banderas se emplean otras representaciones.

La separación entre palabras se indica mediante un espacio, muy simple, la ausencia de un punto o un guión. En los sistemas basados en sonido, un espacio se indica mediante un espacio corto de tiempo durante el cual no se transmite sonido. En la figura 8.39 mostramos la versión internacional de la clave Morse.

| Carácter | Código | Carácter | Código |
|---|---|---|---|
| A | .- | T | - |
| B | -... | U | ..- |
| C | -.-. | V | ...- |
| D | -.. | W | .-- |
| E | . | X | -..- |
| F | ..-. | Y | -.-- |
| G | --. | Z | --.. |
| H | .... | | |
| I | .. | *Dígitos* | |
| J | .--- | 1 | .---- |
| K | -.- | 2 | ..--- |
| L | .-.. | 3 | ...-- |
| M | -- | 4 | ....- |
| N | -. | 5 | ..... |
| O | --- | 6 | -.... |
| P | .--. | 7 | --... |
| Q | --.- | 8 | ---.. |
| R | .-. | 9 | ----. |
| S | ... | 0 | ----- |

**Figura 8.39** Las letras del alfabeto expresadas en la clave Morse internacional.

Escriba un programa que lea una frase en español y la convierta a clave Morse. Además, escriba un programa que lea la frase en clave Morse y la convierta a su equivalente en español. Utilice un espacio en blanco entre cada letra en clave Morse y tres espacios en blanco entre cada palabra en clave Morse.

**8.40** (*Programa de conversión de medidas.*) Escriba un programa que ayude al usuario a convertir medidas. Su programa debe permitir al usuario especificar los nombres de las unidades como cadenas (es decir, centímetros, litros, gramos, …, para el sistema métrico y, pulgadas, cuartos, libras, …, para el sistema inglés) y debe responder a preguntas simples como

```
"¿Cuántas pulgadas hay en 2 metros?"
"¿Cuántos litros hay en 10 cuartos?"
```

Su programa debe reconocer las conversiones inválidas. Por ejemplo, la pregunta

```
"¿Cuántos pies hay en 5 kilogramos?"
```

no tiene sentido, ya que los **"pies"** son medidas de longitud mientras que los **"kilogramos"** son unidades de masa.

**8.41** (*Cartas para exigir el pago de una deuda*.) Muchas empresas gastan una gran cantidad de tiempo y dinero recuperando deudas atrasadas. *Dunning* es el proceso de solicitar repetida e insistentemente a un deudor que pague su deuda.

A menudo se utilizan las computadoras para generar cartas automáticamente y en grados crecientes de severidad al hacerse vieja una deuda. La teoría es que al hacerse vieja una deuda, se hace más difícil de recuperar, y por lo tanto las cartas para recuperación se hacen más agresivas.

Escriba un programa que contenga el texto de cinco cartas para recuperación cada vez más agresivas. Su programa debe aceptar como entrada lo siguiente:

a) Nombre del deudor.
b) Dirección del deudor.
c) Número de cuenta del deudor.

d) Monto de la deuda.
e) Tiempo del monto de la deuda (es decir, un mes de retraso, dos meses de retraso, etcétera)
Utilice el tiempo de la deuda para seleccionar uno de los cinco mensajes de texto, e imprima la carta de recuperación apropiada, de acuerdo con los datos proporcionados.

## *UN DESAFIANTE PROYECTO DE MANIPUACIÓN DE CADENAS*

**8.42** (*Generador de crucigramas*.) La mayoría de la gente ha resuelto un crucigrama en algún momento de su vida, pero pocos han intentado generar uno. Generar un crucigrama es un problema difícil. Lo sugerimos aquí como un proyecto de manipulación de cadenas que requiere de una sofisticación y esfuerzo importante. Existen muchos aspectos que el programador debe resolver para lograr que incluso el generador de crucigramas más sencillo funcione. Por ejemplo, ¿cómo representar las celdas del crucigrama dentro de la computadora? ¿Se deben utilizar una serie de cadenas, o de arreglos con dos subíndices? El programador necesita una serie de palabras (es decir, un diccionario computarizado) al que se pueda hacer referencia de manera directa en el programa. ¿De qué manera se deben almacenar estas palabras para facilitar las complejas manipulaciones que requiere el programa? El lector en verdad ambicioso querrá generar la parte de las “claves” del crucigrama en la que se imprimen las breves pistas para cada palabra “horizontal” y “vertical”, para quien resuelve el crucigrama. La simple impresión de una versión en blanco del crucigrama no es un problema sencillo.



# Entrada/Salida con formato en C

## Objetivos

- Comprender los flujos de entrada y de salida.
- Utilizar todas las capacidades para formato de impresión.
- Utilizar todas las capacidades para formato de entrada.
- Impresión con longitudes de campo y precisiones.
- Utilizar banderas de formato en la cadena de control de formato de **`printf`**.
- Desplegar literales y secuencias de escape.

*Todas las noticias que vale la pena imprimir*.
Adolph S. Ochs

*¿Qué loca búsqueda? ¿Qué lucha para escapar?*
John Keats

*No remuevas las marcas en los límites de los campos*.
Amenemope



## Plan general



## 9.1 Introducción

Una parte importante de la solución de cualquier problema es la presentación de los resultados. En este capítulo, explicaremos con profundidad las características de formato de ***scanf*** y ***printf***. Estas funciones introducen datos desde el *flujo estándar de entrada* y arrojan los datos al *flujo estándar de salida*, respectivamente. En el capítulo 8, explicamos otras cuatro funciones que utilizan la entrada y la salida estándar: **gets**, **puts**, **getchar** y **putchar**. Incluya el encabezado **<stdio.h>** en programas que llamen a estas funciones.

Anteriormente explicamos muchas de las características de **printf** y **scanf**. Este capítulo resume estas características e introduce otras. El capítulo 11 explica muchas funciones adicionales incluidas en la biblioteca estándar de entrada/salida (**stdio**).

## 9.2 Flujos

Toda la entrada y salida se realiza por medio de *flujos*, los cuales son secuencias de bytes. En operaciones de entrada, los bytes fluyen desde un dispositivo (por ejemlo, el teclado, el disco duro, una conexión de red) hacia la memoria principal. En operaciones de salida, los bytes fluyen desde la memoria principal hacia un dispositivo (por ejemplo, una pantalla, una impresora, un disco duro, una conexión de red, etcétera).

Cuando comienza la ejecución del programa, automáticamente se conectan tres flujos al programa. Por lo general, el flujo estándar de entrada se conecta al teclado y el flujo estándar de salida se conecta a la pantalla. A menudo, los sistemas operativos permiten *redireccionar* estos flujos hacia otros dispositivos. Un tercer flujo, el *flujo estándar de error*, se conecta a la pantalla. Los mensajes de error se arrojan al flujo estándar de error. Explicaremos con detalle los flujos en el capítulo 11, Procesamiento de archivos en C.

## 9.3 Formato de salida con printf

El formato preciso de salida se logra con la instrucción **printf**. Cada llamada a **printf** contiene una *cadena de control de formato* que describe el formato de salida. La cadena de control de formato consta de *especificadores de conversión*, *banderas*, *anchos de campo*, *precisiones* y *literales de carácter*. Juntos con el signo de porcentaje (**%**) forman las *especificaciones de conversión*. La función **printf** tiene las siguientes capacidades de formato, cada una de las cuales explicaremos en este capítulo.

1. *Redondeo* de valores de punto flotante hasta un número indicado de posiciones decimales.
2. *Alineación* de una columna de número con puntos decimales que aparecen uno sobre el otro.

3. *Justificación a la izquierda* y *justificación a la derecha* de resultados.
4. *Inserción de literales de carácter* en la ubicación precisa de una línea de salida.
5. Representación de *números de punto flotante* en formato exponencial.
6. Representación de *enteros sin signo* en formato octal y decimal. Vea el apéndice E para mayor información respecto a los valores octales y hexadecimales.
7. Desplegado de todos los tipos de datos con anchos de campo y precisiones fijas.

La función **`printf`** tiene la forma

**`printf(`** *cadena de control de formato, otros argumentos* **`);`**

la *cadena de control de formato* describe el formato de salida y, *otros argumentos* (los cuales son opcionales) corresponden a cada especificación de conversión de la *cadena de control de formato*. Cada especificación de conversión comienza con un signo de porcentaje que termina con un especificador de conversión. Puede haber muchas especificaciones de conversión en una cadena de control de formato.



**Error común de programación 9.1**

*Olvidar encerrar una cadena de control de formato entre comillas, es un error de sintaxis.*



**Buena práctica de programación 9.1**

*Por presentación, edite de manera clara las salidas de un programa, para hacer que éstas sean más legibles y para reducir los errores de usuario.*

## 9.4 Impresión de enteros

Un entero es un número completo, tal como **`776`**, **`0`**, o **`-52`**, que no contiene punto decimal. Los valores enteros se despliegan en uno de varios formatos. La figura 9.1 describe los *especificadores de conversión entera*.

La figura 9.2 imprime un entero por medio de cada uno de los especificadores de conversión. Observe que solamente se imprime el signo menos; el signo más se suprime. Más adelante en el capítulo, veremos cómo forzar la impresión del signo más. También observe que cuado se lee el valor **`-455`** con **`%u`**, éste se convierte al valor sin signo **`4294966841`**.

| Especificador de conversión | Descripción |
|---|---|
| **`d`** | Despliega un entero decimal con signo. |
| **`i`** | Despliega un entero decimal con signo. [*Nota:* Los especificadores **`i`** y **`d`** son diferentes cuando se utilizan con **`scanf`**.] |
| **`o`** | Despliega un entero octal sin signo. |
| **`u`** | Despliega un entero decimal sin signo. |
| **`x`** o **`X`** | Despliega un entero hexadecimal sin signo. **`X`** provoca que se desplieguen los dígitos de **`0`** a **`9`** y las letras de **`A`** a **`F`**, y **`x`** provoca que se desplieguen los dígitos de **`0`** a **`9`** y las letras de **`a`** a **`f`**. |
| **`h`** o **`l`** (letra **`l`**) | Se coloca antes de cualquier especificador de conversión entera para indicar que se despliega un entero corto o largo, respectivamente. Las letras **`h`** y **`l`** son llamadas con más precisión *modificadores de longitud*. |

**Figura 9.1** Especificadores de conversión entera.

```
/* Figura 9.2: fig09_02.c */
/* Uso de los especificadores de conversión entera */
#include <stdio.h>
```

**Figura 9.2** Uso de los especificadores de conversión entera. (Parte 1 de 2.)

```
int main()
{
   printf( "%d\n", 455 );
   printf( "%i\n", 455 );  /* i es lo mismo que d en printf */
   printf( "%d\n", +455 );
   printf( "%d\n", -455 );
   printf( "%hd\n", 32000 );
   printf( "%ld\n", 2000000000 );
   printf( "%o\n", 455 );
   printf( "%u\n", 455 );
   printf( "%u\n", -455 );
   printf( "%x\n", 455 );
   printf( "%X\n", 455 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

**Figura 9.2** Uso de los especificadores de conversión entera. (Parte 2 de 2.)



**Error común de programación 9.2**

*Imprimir un valor negativo con un especificador de conversión que espera un valor* **`unsigned`**.

## 9.5 Impresión de números de punto flotante

Un valor de punto flotante contiene un punto decimal como en **`33.5`**, **`0.0`**, o **`-657.983`**. Los valores de punto flotante se despliegan en uno de varios formatos. La figura 9.3 describe los *especificadores de conversión de punto flotante*. Los especificadores ***`e`*** y ***`E`*** despliegan valores de punto flotante con *notación exponencial*. La notación exponencial es el equivalente en la computadora a la *notación científica* que se utiliza en matemáticas. Por ejemplo, el valor 150.4582 se representa en notación científica como

**1.504582 X** $10^2$

y en la computadora, se representa en notación exponencial como

**`1.504582E+02`**

Esta notación indica que **`1.594582`** se multiplica por **`10`** elevado a la segunda potencia (**`E+02`**). La **`E`** significa "exponente".

Los valores impresos con los especificadores de conversión **`e`**, **`E`** y **`f`** se despliegan de manera predeterminada con una precisión de seis dígitos a la derecha del punto decimal (por ejemplo, 1.04592); se pueden especificar explícitamente otras precisiones. El *especificador de conversión* ***`f`*** siempre imprime al menos un dígito a la izquierda del punto decimal. Los especificadores de conversión **`e`** y **`E`** imprimen, respectivamente, la letra

| Especificador de conversión | Descripción |
|---|---|
| **`e`** o **`E`** | Despliega un valor de punto flotante con notación exponencial. |
| **`f`** | Despliega un valor de punto flotante con notación de punto fijo. |
| **`g`** o **`G`** | Despliega un valor de punto flotante con el formato de punto flotante **`f`**, o con el formato exponencial **`e`** (o **`E`**) basado en la magnitud del valor. |
| **`L`** | Se coloca antes del especificador de conversión para indicar que se desplegará un valor de punto flotante **`long double`**. |

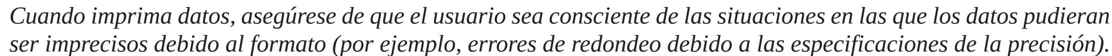
**Figura 9.3** Especificadores de conversión de punto flotante.

minúscula **`e`** o la letra mayúscula **`E`** que precede al exponente, y siempre imprimen exactamente un dígito a la izquierda del punto decimal.

El *especificador de conversión* ***`g`*** *(o* ***`G`****)* imprime ya sea una **`e`** (**`E`**), o el formato **`f`** sin acarreo de ceros a la derecha (por ejemplo, **`1.234000`** se imprime como **`1.234`**). Los valores se imprimen con **`e`** (**`E`**) si, después de convertir el valor a la notación exponencial, el valor del exponente es menor que **`-4`**, o el exponente es mayor o igual que la precisión especificada (seis dígitos significativos de manera predeterminada para **`g`** o **`G`**). De lo contrario, se utiliza el especificador de conversión **`f`** para imprimir el valor. Con **`g`** o **`G`**, los ceros de acarreo no se imprimen en la parte fraccionaria del valor de salida. Se requiere al menos un dígito decimal para la impresión del punto decimal. Con el especificador de conversión **`g`**, los valores **`0.0000875`**, **`8750000.0`**, **`8.75`**, **`87.50`** y **`875`** se imprimen como **`8.75e-05`**, **`8.75e+06`**, **`8.75`**, **`87.5`** y **`875`**. El valor **`0.0000875`** utiliza la notación **`e`** debido a que, cuando se convierte a la notación exponencial, su exponente (**`-5`**) es menor que **`-4`**. El valor **`8750000.0`** utiliza la notación **`e`**, debido a que su exponente (**`6`**) es igual que la precisión predeterminada.

La precisión para los especificadores de conversión **`g`** y **`G`** indican el número máximo de dígitos significativos que se imprimen, incluyendo el dígito a la derecha del punto decimal. El valor **`1234567.0`** se imprime como **`1.23457e+06`**, con el uso del especificador de conversión **`%g`** (recuerde que todos los especificadores de conversión de punto flotante tienen una precisión predeterminada de 6). Observe que existen 6 dígitos significativos en el resultado. La diferencia entre **`g`** y **`G`** es idéntica a la diferencia entre **`e`** y **`E`** cuando el valor se imprime mediante notación exponencial; la letra minúscula **`g`** provoca la salida de una letra minúscula **`e`**, y la letra mayúscula **`G`** provoca la salida de la letra mayúscula **`E`**.

### Tip para prevenir errores 9.1



*Cuando imprima datos, asegúrese de que el usuario sea consciente de las situaciones en las que los datos pudieran ser imprecisos debido al formato (por ejemplo, errores de redondeo debido a las especificaciones de la precisión).*

La figura 9.4 muestra cada uno de los especificadores de conversión de punto flotante. Observe que los especificadores de conversión **`%E`**, **`%e`** y **`%g`** provocan el redondeo del valor de salida, no así el especificador de conversión **`%f`**.

```
/* Figura 9.4: fig09_04.c */
/* Impresión de números de punto flotante con
   especificadores de conversión de punto flotante */

#include <stdio.h>

int main()
{
   printf( "%e\n", 1234567.89 );
   printf( "%e\n", +1234567.89 );
   printf( "%e\n", -1234567.89 );
```

**Figura 9.4** Uso de los especificadores de conversión de punto flotante. (Parte 1 de 2.)

```
    printf( "%E\n", 1234567.89 );
    printf( "%f\n", 1234567.89 );
    printf( "%g\n", 1234567.89 );
    printf( "%G\n", 1234567.89 );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

**Figura 9.4** Uso de los especificadores de conversión de punto flotante. (Parte 2 de 2.)

## 9.6 Impresión de cadenas y caracteres

Los especificadores de conversión **c** y **s** se utilizan para imprimir caracteres individuales y cadenas, respectivamente. El *especificador de conversión* ***c*** requiere un argumento **char**. El *especificador de conversión* ***s*** requiere como argumento un apuntador a **char**. El especificador de conversión **s** provoca la impresión de los caracteres hasta que encuentra el carácter de terminación nulo (**'\0'**). El programa que muestra la figura 9.5 despliega los caracteres y las cadenas con los especificadores de conversión **c** y **s**.

```
/* Figura 9.5: fig09_05c */
/* Impresión de cadenas y caracteres */
#include <stdio.h>

int main()
{
   char caracter = 'A'; /* inicializa un char */
   char cadena[] = "Esta es una cadena"; /* inicializa el arreglo char */
   const char *ptrCadena = "Esta tambien es una cadena"; /* apuntador a char */

   printf( "%c\n", caracter );
   printf( "%s\n", "Esta es una cadena" );
   printf( "%s\n", cadena );
   printf( "%s\n", ptrCadena );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
A
Esta es una cadena
Esta es una cadena
Esta tambien es una cadena
```

**Figura 9.5** Uso de los especificadores de conversión para caracteres y cadenas.



### Error común de programación 9.3

*Utilizar un* **`%c`** *para imprimir una cadena es un error. El especificador de conversión* **`%c`** *espera un* **`char`** *como argumento. Una cadena es un apuntador a* **`char`** *(es decir, un* **`char *`***).*



### Error común de programación 9.4

*En algunos sistemas, utilizar un* **`%s`** *para imprimir un argumento* **`char`***, provoca un error fatal en tiempo de ejecución llamado violación de acceso. El especificador de conversión* **`%s`** *espera un argumento de tipo apuntador a* **`char`***.*



### Error común de programación 9.5

*Utilizar comillas sencillas alrededor de cadenas de caracteres es un error de sintaxis. Las cadenas de caracteres deben encerrarse entre comillas dobles.*



### Error común de programación 9.6

*Utilizar comillas dobles alrededor de una constante de carácter crea una cadena que consiste en dos caracteres, en la cual el segundo carácter es el nulo de terminación. Una constante de carácter es un carácter individual encerrado entre comillas sencillas.*

## 9.7 Otros especificadotes de conversión

Los tres especificadores de conversión restantes son **p**, **n** y **%** (figura 9.6).



### Tip de portabilidad 9.1

*El especificador de conversión* **`p`** *despliega una dirección de manera definida en la implementación (en muchos sistemas, se utiliza la notación hexadecimal en lugar de la notación decimal).*

El *especificador de conversión* **`n`** almacena el número de caracteres ya impresos con la instrucción **`printf`** actual, el argumento correspondiente es un apuntador a una variable entera, en la cual se almacena el valor. El especificador de conversión **`%n`** no imprime valor alguno. El especificador de conversión **`%`** provoca la salida de un signo de porcentaje.

El **`%p`** de la figura 9.7 imprime el valor de **`ptr`** y la dirección de **`x`**; estos valores son idénticos debido a que a **`ptr`** se le asigna la dirección de **`x`**. A continuación, **`%n`** almacena el número de caracteres de salida de la tercera instrucción **`printf`** (línea 15) en la variable entera **`y`**, e imprime el valor de **`y`**. La ultima instrucción

| Especificador de conversión | Descripción |
|---|---|
| **p** | Despliega un valor apuntador de manera definida por la implementación. |
| **n** | Almacena el número de caracteres ya desplegados en la instrucción **`printf`** actual. Proporciona un apuntador a un entero como el argumento correspondiente. No despliega valor alguno. |
| **%** | Despliega el carácter de porcentaje. |

**Figura 9.6** Otros especificadores de conversión.

```
1  /* Figura 9.7: fig09_07.c */
2  /* Uso de los especificadores de conversión p, n, y % */
3  #include <stdio.h>
4
5  int main()
6  {
7     int *ptr;      /* define un apuntador a un int */
8     int x = 12345; /* inicializa int x */
9     int y;         /* define int y */
```

**Figura 9.7** Uso de los especificadores de conversión **p**, **n** y **%.** (Parte 1 de 2.)

```
10
11     ptr = &x;      /* asigna a ptr la dirección de x */
12     printf( “El valor de ptr es %p\n”, ptr );
13     printf( “La direccion de x es %p\n\n”, &x );
14
15     printf( “Total de caracteres impresos en esta linea:%n”, &y );
16     printf( “ %d\n\n”, y );
17
18     y = printf( “Esta linea tiene 30 caracteres\n” );
19     printf( “ se imprimieron %d caracteres\n\n”, y );
20
21     printf( “Impresion de %% en una cadena de control de formato\n” );
22
23     return 0; /* indica terminación exitosa */
24
25  } /* fin de main */
```

```
El valor de ptr es 0012FF78
La direccion de x es 0012FF78

Total de caracteres impresos en esta linea: 43

Esta linea tiene 30 caracteres
se imprimieron 31 caracteres

Impresion de % en una cadena de control de formato
```

**Figura 9.7** Uso de los especificadores de conversión **p**, **n** y **%**. (Parte 2 de 2.)

**`printf`** (línea 21) utiliza **%%** para imprimir el carácter **%** en la cadena de caracteres. Observe que cada llamada a **`printf`** devuelve un valor, ya sea el número de caracteres de salida, o un valor negativo si ocurre un error en la salida.



### Error común de programación 9.7

*Intentar imprimir una literal del carácter de porcentaje mediante el uso de **%** en lugar de **%%** dentro de la cadena de control de formato, es un error. Cuando aparece **%** en una cadena de control de formato, debe ser seguida por un especificador de conversión.*

## 9.8 Impresión con ancho de campos y precisiones

El tamaño exacto de un campo en el que se imprimen datos se especifica por medio del *ancho de campo*. Si el ancho del campo es mayor que el dato a imprimir, por lo general el dato se justifica a la derecha dentro del campo. El entero que representa el ancho del campo se inserta entre el signo de porcentaje (**%**) y el especificador de conversión (por ejemplo, **%4d**). La figura 9.8 imprime dos grupos de cinco números cada uno, y justifica a la derecha aquellos campos que contienen menos dígitos que el ancho del campo. Observe que el ancho del campo se incrementa para imprimir los valores más grandes que el campo, y que el signo menos para los valores negativos utiliza solamente una posición de carácter en el ancho del campo. Los anchos de campo se pueden utilizar con todos los especificadores de conversión.



### Error común de programación 9.8

*No proporcionar un ancho de campo suficiente para manipular un valor de impresión puede ocasionar el desplazamiento de otros datos en la impresión y producir salidas confusas. ¡Conozca sus datos!*

La función **`printf`** también proporciona la habilidad para especificar la precisión con la que se imprimen los datos. La precisión tiene significados diferentes para diferentes tipos de datos. Cuando se utilizan con es-

```
/* Figura 9.8: fig09_08.c */
/* Impresión de enteros justificados a la derecha */
#include <stdio.h>

int main()
{
   printf( "%4d\n", 1 );
   printf( "%4d\n", 12 );
   printf( "%4d\n", 123 );
   printf( "%4d\n", 1234 );
   printf( "%4d\n\n", 12345 ); /* dato demasiado largo */

   printf( "%4d\n", -1 );
   printf( "%4d\n", -12 );
   printf( "%4d\n", -123 );
   printf( "%4d\n", -1234 );   /* dato demasiado largo */
   printf( "%4d\n", -12345 );  /* dato demasiado largo */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

**Figura 9.8** Justificación derecha de enteros dentro de un campo.

pecificadores de conversión entera, la precisión indica el número mínimo de dígitos a desplegar. Si el valor impreso contiene menos dígitos que la precisión especificada, se colocan ceros como prefijo hasta que el número total de dígitos es equivalente a la precisión. La precisión predeterminada para los enteros es **1**. Cuando se utiliza con los especificadores de conversión de punto flotante **e**, **E** y **f**, la precisión es el número de dígitos que aparecen después del punto decimal. Cuado se utiliza con los especificadores de conversión **g** y **G**, la precisión es el máximo número de dígitos significativos que se van a imprimir. Cuado se utiliza con el especificador de conversión **s**, la precisión es el máximo número de caracteres a escribir en la cadena. Para utilizar la precisión, coloque un punto decimal (**.**), seguido por un carácter entre el signo de porcentaje y el especificador de conversión que representa la precisión. La figura 9.9 muestra el uso de la precisión dentro de las cadenas de control de formato. Observe que cuando un valor de punto flotante se imprime con una precisión menor que el número original de posiciones decimales, el valor se redondea.

El ancho de campo y la precisión pueden combinarse, colocando el ancho del campo, seguido por un punto decimal, seguido por la precisión, entre el signo de porcentaje y el especificador de conversión, como en la instrucción

```
printf( "%9.3f", 123.456789 );
```

la cual despliega **123.457** con tres dígitos a la derecha del punto decimal, justificado a la derecha en un campo de nueve posiciones.

```
/* Figura 9.9: fig09_09.c */
/* Uso de la precisión durante la impresión de enteros,
   números de punto flotante, y cadenas */
#include <stdio.h>

int main()
{
   int i = 873;                    /* inicializa el entero int i */
   double f = 123.94536;           /* inicializa el double f */
   char s[] = “Feliz Cumpleanios”; /* inicializa el arreglo char s */

   printf( “Uso de la precision en enteros\n” );
   printf( “\t%.4d\n\t%.9d\n\n”, i, i );

   printf( “Uso de la precision en numeros de punto flotante\n” );
   printf( “\t%.3f\n\t%.3e\n\t%.3g\n\n”, f, f, f );

   printf( “Uso de la precision en cadenas\n” );
   printf( “\t%.11s\n”, s );

   return 0; /* indica terminacion exitosa */

} /* fin de main */
```

```
Uso de la precision en enteros
        0873
        000000873

Uso de la precision en numeros de punto flotante
        123.945
        1.239e+002
        124

Uso de la precision en cadenas
        Feliz Cumpleanios
```

**Figura 9.9** Uso de la precisión para desplegar información de varios tipos.

Es posible especificar el ancho del campo y la precisión mediante expresiones enteras en la lista de argumentos después de la cadena de control de formato. Para utilizar esta característica, inserte un asterisco (**`*`**) en lugar del ancho del campo o de la precisión (o ambos). El argumento **`int`** que coincide con la lista de argumentos se evalúa y se utiliza en lugar del asterisco. El valor del ancho de un campo puede ser positivo o negativo (lo cual provoca que la salida se justifique a la izquierda o a la derecha, como explicaremos en la siguiente sección). La instrucción

```
printf( “%*.*f”, 7, 2, 98.736 );
```

utiliza **`7`** para el ancho del campo, **`2`** para la precisión, e imprime el valor **`98.74`** justificado a la derecha.

## 9.9 Uso de banderas en la cadena de control de formato de `printf`

La función **`printf`** también proporciona banderas para complementar las capacidades de formato de las salidas. Existen cinco banderas disponibles para utilizarlas dentro de las cadenas de control de formato (figura 9.10).

Para utilizar una bandera dentro de una cadena de control de formato, coloque la bandera inmediatamente a la derecha del signo de porcentaje. Se pueden combinar varias banderas en un sólo especificador de conversión.

| Bandera | Descripción |
|---|---|
| − (signo menos) | Justifica la salida a la izquierda dentro del campo especificado. |
| + (signo más) | Despliega el signo más que precede a los valores positivos, y un signo menos que precede a los valores negativos. |
| *espacio* | Imprime un espacio antes de un valor positivo no impreso con la bandera +. |
| # | Prefijo **0** para el valor de salida utilizado con el especificador de conversión octal **o**. |
| | Prefijo **0x** o **0X** para el valor de salida cuando se utiliza con el especificador de conversión de formato **x** o **X**. |
| | Fuerza la impresión del punto decimal de un número de punto flotante impreso con **e**, **E**, **f**, **g** o **G** que no contiene una parte fraccionaria. (Por lo general, el punto decimal solamente se imprime si le sigue un dígito.) Para los especificadores **g** y **G**, no se eliminan los ceros de acarreo. |
| 0 (cero) | Rellena con ceros el principio de un campo. |

**Figura 9.10** Banderas de la cadena de control de formato.

La figura 9.11 muestra la justificación derecha e izquierda de una cadena, un entero, un carácter y un número de punto flotante.

```
/* Figura 9.11: fig09_11.c */
/* Justificación derecha e izquierda de valores */
#include <stdio.h>

int main()
{
   printf( "%10s%10d%10c%10f\n\n", "hola", 7, 'a', 1.23 );
   printf( "%-10s%-10d%-10c%-10f\n", "hola", 7, 'a', 1.23 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
      hola         7         a  1.230000

hola      7         a         1.230000
```

**Figura 9.11** Justificación derecha en un campo.

La figura 9.12 imprime un número positivo y un número negativo, cada uno con y sin la *bandera* +. Observe que en ambos casos se despliega el signo menos, pero el signo más solamente se despliega cuando se utiliza la bandera +.

```
/* Figura 9.12: fig09_12.c */
/* Impresión de números con y sin la bandera + */
#include <stdio.h>

int main()
{
   printf( "%d\n%d\n", 786, -786 );
   printf( "%+d\n%+d\n", 786, -786 );

```

**Figura 9.12** Impresión de números positivos y negativos con y sin la bandera +. (Parte 1 de 2.)

```
    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
786
-786
+786
-786
```

**Figura 9.12** Impresión de números positivos y negativos con y sin la bandera +. (Parte 2 de 2.)

La figura 9.13 coloca el espacio como prefijo de un número positivo con la *bandera espacio*. Esto es útil para alinear los números positivos y negativos con el mismo número de dígitos. Observe que al valor **`-547`** no le precede un espacio en la salida, debido a que tiene un signo menos.

```
/* Figura 9.13: fig09_13.c */
/* Impresión de un espacio antes de los valores con signo
   no precedidos por + o - */
#include <stdio.h>

int main()
{
   printf( “% d\n% d\n”, 547, -547 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
547
-547
```

**Figura 9.13** Uso de la bandera espacio.

La figura 9.14 utiliza la *bandera #* como prefijo de **`0`** para un valor octal, y **`0x`** y **`0X`** para los valores hexadecimales, y fuerza al punto decimal con un valor impreso con **`g`**.

```
/* Figura 9.14: fig09_14.c */
/* Uso de la bandera # con los especificadores de conversión
   o, x, X y cualquier especificador de punto flotante */
#include <stdio.h>

int main()
{
   int c = 1427;       /* inicializa c */
   double p = 1427.0; /* inicializa p */

   printf( “%#o\n”, c );
   printf( “%#x\n”, c );
   printf( “%#X\n”, c );
   printf( “\n%g\n”, p );
   printf( “%#g\n”, p );
```

**Figura 9.14** Uso de la bandera #. (Parte 1 de 2.)

```

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
02623
0x593
0X593

1427
1427.00
```

**Figura 9.14** Uso de la bandera #. (Parte 2 de 2.)

La figura 9.15 combina la bandera + y la bandera **`0`** (cero) para imprimir **`452`** y un campo de **`9`** posiciones con un signo + y ceros al inicio; posteriormente imprime de nuevo **`452`** utilizando sólo la bandera **`0`** y un campo de **`9`** posiciones.

```
/* Figura 9.15: fig09_15.c */
/* La impresión con la bandera 0( cero ) llena con ceros el inicio de un
   campo*/
#include <stdio.h>

int main()
{
   printf( “%+09d\n”, 452 );
   printf( “%09d\n”, 452 );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
+00000452
000000452
```

**Figura 9.15** Uso de la bandera 0 (cero).

## 9.10 Impresión de literales y secuencias de escape

La mayoría de las literales de carácter que se imprimen con **`printf`** simplemente pueden incluirse en la cadena de control de formato. Sin embargo, existen varios caracteres "problemáticos", tales como las comillas (**"**), que delimitan la propia cadena de control de formato. Varios caracteres de control, tales como una nueva línea y el tabulador, deben representarse como una *secuencia de escape*. Una secuencia de escape se representa con una diagonal invertida (****), seguida por un *carácter de escape* en particular. La figura 9.16 lista las secuencias de escape y las acciones que provocan.

| Secuencia de escape | Descripción |
|---|---|
| **\\'** (comilla sencilla) | Despliega el carácter de comilla sencilla (**'**). |
| **\\"** (comilla doble) | Despliega el carácter de comilla doble (**"**). |

**Figura 9.16** Secuencias de escape. (Parte 1 de 2.)

| Secuencia de escape | Descripción |
|---|---|
| `\?` (interrogación) | Despliega el carácter de signo de interrogación (`?`). |
| `\\` (diagonal invertida) | Despliega el carácter de diagonal invertida (`\`). |
| `\a` (alerta o campana) | Provoca una alerta sonora (campana) o una alerta visual. |
| `\b` (retroceso) | Mueve el cursor una posición hacia atrás en la línea actual. |
| `\f` (nueva página o avance de página) | Mueve el cursor al inicio de la siguiente página lógica. |
| `\n` (nueva línea) | Mueve el cursor al principio de la siguiente línea. |
| `\r` (retorno de carro) | Mueve el cursor al principio de la línea actual. |
| `\t` (tabulador horizontal) | Mueve el cursor a la siguiente posición del tabulador horizontal. |
| `\v` (tabulador vertical) | Mueve el cursor a la siguiente posición del tabulador vertical. |

**Figura 9.16** Secuencias de escape. (Parte 2 de 2.)



**Error común de programación 9.9**

*Intentar imprimir una comilla sencilla, una comilla doble, un signo de interrogación o una diagonal invertida como un dato literal dentro de una instrucción **`printf`**, sin colocar una diagonal invertida para formar una secuencia de escape, es un error de sintaxis.*

## 9.11 Formato de entrada con `scanf`

El formato preciso de entrada se puede lograr con **`scanf`**. Cada instrucción **`scanf`** contiene una cadena de control de formato que describe el formato de los datos de entrada. La cadena de control de formato consta de especificadores de conversión y literales de cadena. La función **`scanf`** tiene las siguientes capacidades de formato:

1. Introduce todo tipo de datos.
2. Introduce caracteres específicos desde un flujo de entrada.
3. Ignora caracteres específicos del flujo de entrada.

La función **`scanf`** se escribe de la siguiente manera:

**`scanf (`** *cadena de control de formato, otros argumentos* **`);`**

La *cadena de control de formato* describe los formatos de la entrada, y *otros argumentos* son apuntadores a las variables en las que se almacenará la entrada.



**Buena práctica de programación 9.2**

*Cuando se introduzcan datos, solicite al usuario uno o varios elementos a la vez. Evite pedir al usuario que introduzca muchos elementos en respuesta a una sola indicación.*

La figura 9.17 resume los especificadores de conversión utilizados para imprimir todos los tipos de datos. El resto de esta sección proporciona programas para demostrar la lectura de datos con los distintos especificadores de conversión de **`scanf`**.

| Especificador de conversión | Descripción |
|---|---|
| *Enteros* | |
| **`d`** | Lee un entero decimal con signo (el signo es opcional). El argumento correspondiente es un apuntador a un entero. |
| **`i`** | Lee un entero decimal, octal, o hexadecimal con signo (opcional). El argumento correspondiente es un apuntador a un entero. |

**Figura 9.17** Especificadores de conversión para **`scanf`**. (Parte 1 de 2.)

| Especificador de conversión | Descripción |
|---|---|
| **o** | Lee un entero octal. El argumento correspondiente es un apuntador a un entero sin signo. |
| **u** | Lee un entero decimal sin signo. El argumento correspondiente es un apuntador a un entero sin signo. |
| **x** o **X** | Lee un entero hexadecimal. El argumento correspondiente es un apuntador a un entero sin signo. |
| **h** o **l** | Se coloca antes de cualquier especificador de conversión, para indicar que se introducirá un entero corto o largo, respectivamente. |
| *Números de punto flotante* | |
| **e**, **E**, **f**, **g** o **G** | Lee un valor de punto flotante. El argumento correspondiente es un apuntador a un valor de punto flotante. |
| **l** o **L** | Se coloca antes de cualquier especificador de conversión, para indicar que se introducirá un valor **double** o **long double**. El argumento correspondiente es un apuntador a una variable **double** o **long double**. |
| *Cadenas y caracteres* | |
| **c** | Lee un carácter. El argumento correspondiente es un apuntador a **char**; no agrega el carácter nulo (**'\0'**). |
| **s** | Lee una cadena. El argumento correspondiente es un apuntador a un arreglo de tipo **char** que sea lo suficientemente grande para almacenar la cadena y el carácter nulo (**'\0'**), el cual se agrega automáticamente. |
| *Conjunto de exploración* | |
| *[caracteres de exploración]* | Busca en una cadena un conjunto de caracteres almacenados en un arreglo. |
| *Varios* | |
| **p** | Lee una dirección de la misma forma que la dirección de salida con **%p** dentro de una instrucción **printf**. |
| **n** | Almacena el número de caracteres de entrada de **scanf**. El argumento correspondiente es un apuntador a un entero. |
| **%** | Ignora el signo de porcentaje en la entrada. |

**Figura 9.17** Especificadores de conversión para **scanf**. (Parte 2 de 2.)

La figura 9.18 lee enteros con los distintos especificadores de conversión y despliega los enteros como números decimales. Observe que **%i** es capaz de introducir enteros decimales, octales y hexadecimales.

```
/* Figura 9.18: fig09_18.c */
/* Lectura de enteros */
#include <stdio.h>

int main()
{
   int a;
   int b;
   int c;
   int d;
   int e;
   int f;
```

**Figura 9.18** Lectura de enteros mediante especificadores de conversión entera. (Parte 1 de 2.)

```
    int g;

    printf( "Introduzca siete enteros: " );
    scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

    printf( "La entrada desplegada como enteros decimales es:\n" );
    printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca siete enteros: -70 -70 070 0x70 70 70 70
La entrada desplegada como enteros decimales es:
-70 -70 56 112 56 70 112
```

**Figura 9.18** Lectura de enteros mediante especificadores de conversión entera. (Parte 2 de 2.)

Cuando se introducen números de punto flotante, es posible utilizar cualquiera de los especificadores de punto flotante **e**, **E**, **f**, **g** o **G**. La figura 9.19 lee tres números de punto flotante, con cada uno de los tres tipos de especificadores de conversión, y despliega los tres números con el especificador de conversión **f**. Observe que la salida del programa confirma el hecho de que los valores de punto flotante son imprecisos; este hecho se resalta en el tercer número impreso.

```
/* Figura 9.19: fig09_19.c */
/* Lectura de números de punto flotante */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
    double a;
    double b;
    double c;

    printf( "Introduzca tres numeros de punto flotante: \n" );
    scanf( "%le%lf%lg", &a, &b, &c );

    printf( "Aqui estan los numeros introducidos con notacion \n" );
    printf( "plana de punto flotante:\n" );
    printf( "%f\n%f\n%f\n", a, b, c );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca tres numeros de punto flotante:
1.27987 1.27987e+03 3.38476e-06
Aqui estan los numeros introducidos con notacion
plana de punto flotante:
1.279870
1279.870000
0.000003
```

**Figura 9.19** Lectura de entradas mediante especificadores de conversión de punto flotante.

Los caracteres y las cadenas se introducen mediante los especificadores de conversión **c** y **s**, respectivamente. La figura 9.20 indica al usuario que introduzca una cadena. El programa introduce el primer carácter de la cadena con **%c** y la almacena en la variable de carácter **x**; luego introduce el resto de la cadena con **%s** y la almacena en el arreglo de caracteres **y**.

```
/* Figura 9.20: fig09_20.c */
/* Lectura de caracteres y cadenas */
#include <stdio.h>

int main()
{
   char x;
   char y[ 9 ];

   printf( “Introduzca una cadena: “ );
   scanf( “%c%s”, &x, y );

   printf( “La entrada fue:\n” );
   printf( “el caracter \”%c\” “, x );
   printf( “y la cadena \”%s\”\n”, y );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca una cadena: Domingo
La entrada fue:
el caracter “D” y la cadena “omingo”
```

**Figura 9.20** Entrada de caracteres y cadenas.

También es posible utilizar el *conjunto de exploración* para introducir una secuencia de caracteres. Un conjunto de exploración es un conjunto de caracteres encerrados entre corchetes, **[]**, y precedidos por el signo de porcentaje en la cadena de control de formato. Un conjunto de exploración examina los caracteres del flujo de entrada, buscando solamente los caracteres que coincidan con los caracteres contenidos en el conjunto de exploración. Cada vez que un carácter coincide, éste se almacena en el argumento correspondiente del conjunto de exploración; un apuntador a un arreglo de caracteres. El conjunto de exploración termina la introducción de caracteres, cuando encuentra un carácter que no está contenido en el conjunto de exploración. Si el primer carácter del flujo de entrada no coincide con un carácter del conjunto de exploración, solamente se almacena el carácter nulo en el arreglo. La figura 9.21 utiliza el conjunto de exploración **[aeiou]** para explorar el flujo de entrada en busca de las vocales. Observe que se leen las primeras siete letras de la entrada. La octava letra (**h**) no se encuentra en el conjunto de exploración y, por lo tanto, termina la exploración.

```
/* Figura 9.21: fig09_21.c */
/* Uso de un conjunto de exploración */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   char z[ 9 ]; /* define el arreglo z */

```

**Figura 9.21** Uso del conjunto de exploración. (Parte 1 de 2.)

```
    printf( “Introduzca una cadena: “ );
    scanf( “%[aeiou]”, z ); /* busca un conjunto de caracteres */

    printf( “La entrada es \”%s\”\n”, z );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca una cadena: ooeeooahah
La entrada es “ooeeooa”
```

**Figura 9.21** Uso del conjunto de exploración. (Parte 2 de 2.)

El conjunto de exploración también puede utilizarse para explorar los caracteres que no están contenidos en el conjunto de exploración por medio de un *conjunto de exploración invertido*. Para crear un conjunto de exploración invertido, coloque una *tilde* (**^**) en los corchetes, antes del conjunto de exploración. Esto provoca que se almacenen los caracteres que no aparecen en el conjunto de exploración. Cuando se encuentra un carácter contenido en el conjunto de exploración invertido, termina la entrada. La figura 9.22 utiliza el conjunto de exploración invertido **[^aeiou]** para la búsqueda de consonantes, o más apropiadamente, para buscar “no vocales”.

```
/* Figura 9.22: fig09_22.c */
/* Uso de un conjunto de exploración invertido */
#include <stdio.h>

int main()
{
    char z[ 9 ];

    printf( “Introduzca una cadena: “ );
    scanf( “%[^aeiou]”, z ); /* conjunto de exploración invertido */

    printf( “La entrada es \”%s\”\n”, z );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca una cadena: Cadena
La entrada es “C”
```

**Figura 9.22** Uso de un conjunto de exploración invertido.

Se puede utilizar el ancho del campo dentro de un especificador de conversión en **scanf**, para leer un número de caracteres desde el flujo de entrada. La figura 9.23 introduce una serie de dígitos consecutivos como dos enteros de dos dígitos y un entero que consiste en el resto de los dígitos del flujo de entrada.

```
/* Figura 9.23: fig09_23.c */
/* entrada de datos con un ancho de campo */
#include <stdio.h>
```

**Figura 9.23** Entrada de datos con un ancho de campo. (Parte 1 de 2.)

```
int main()
{
   int x;
   int y;

   printf( "Introduce un entero de seis digitos: " );
   scanf( "%2d%d", &x, &y );

   printf( "Los enteros introducidos son %d y %d\n", x, y );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduce un entero de seis digitos: 123456
Los enteros introducidos son 12 y 3456
```

**Figura 9.23** Entrada de datos con un ancho de campo. (Parte 2 de 2.)

Con frecuencia, es necesario ignorar ciertos caracteres del flujo de entrada. Por ejemplo, una fecha podría introducirse como

**11-10-1999**

Cada número en la fecha necesita almacenarse, pero pueden descartarse los guiones que separan los números. Para eliminar los caracteres innecesarios, inclúyalos en la cadena de control de formato de **scanf** (los caracteres de *espacio en blanco*, como espacios, nuevas líneas y tabuladores, ignoran todos los espacios en blanco que se encuentran al inicio del campo). Por ejemplo, para ignorar los guiones en la entrada, utilice la instrucción

**scanf( "%d-%d-%d", &mes, &dia, &anio );**

Aunque este **scanf** elimina los guiones de la entrada anterior, es posible introducir la fecha como

**10/11/1999**

En este caso, la instrucción **scanf** anterior no elimina los caracteres innecesarios. Por esta razón, **scanf** proporciona el *carácter de supresión de asignación* *. El carácter de supresión de asignación permite a **scanf** leer cualquier tipo de dato desde la entrada y descartarlo sin asignarlo a una variable. La figura 9.24 utiliza el carácter de supresión de asignación en el especificador de conversión **%c**, para indicar que se debe leer y descartar el carácter que aparece en el flujo de entrada. Solamente se almacenan el mes, el día, y el año. Los valores de las variables se imprimen para demostrar que, de hecho, se introdujeron correctamente. Observe que las listas de argumentos para cada llamada a **scanf** no contienen variables para los especificadores de conversión que utilizan el carácter de supresión de asignación. Los caracteres correspondientes simplemente se descartan.

```
/* Figura 9.24: fig09_24.c */
/* Lectura y descarte de caracteres desde el flujo de entrada */
#include <stdio.h>

int main()
{
   int mes1;  /* define mes1  */
   int dia1;  /* define dia1  */
   int anio1; /* define anio1 */
```

**Figura 9.24** Lectura y descarte de caracteres desde el flujo de entrada. (Parte 1 de 2.)

```
    int mes2;  /* define mes2  */
    int dia2;  /* define dia2  */
    int anio2; /* define anio2 */

    printf( "Introduzca una fecha de la forma mm-dd-aaaa: " );
    scanf( "%d%*c%d%*c%d", &mes1, &dia1, &anio1 );

    printf( "mes = %d  dia = %d  anio = %d\n\n", mes1, dia1, anio1 );

    printf( "Introduzca una fecha de la forma mm/dd/aaaa: " );
    scanf( "%d%*c%d%*c%d", &mes2, &dia2, &anio2 );

    printf( "mes = %d  dia = %d  anio = %d\n", mes2, dia2, anio2 );

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca una fecha de la forma mm-dd-aaaa: 11-18-2003
mes = 11  dia = 18  anio = 2003

Introduzca una fecha de la forma mm/dd/aaaa: 11/18/2003
mes = 11  dia = 18  anio = 2003
```

**Figura 9.24** Lectura y descarte de caracteres del flujo de entrada. (Parte 2 de 2.)

## *RESUMEN*

- Toda entrada y salida de datos se lleva a cabo por medio de flujos, es decir, secuencias de caracteres organizados en líneas. Cada línea consiste en cero o más caracteres y termina con el carácter de nueva línea.
- Por lo general, el flujo estándar de entrada se conecta al teclado, y el flujo estándar de salida se conecta a la pantalla de la computadora.
- A menudo, los sistemas operativos permiten a los flujos estándares de entrada y salida redireccionarse hacia otros dispositivos.
- La cadena de control de formato de **`printf`** describe el formato en el cual aparecerán los valores de salida. La cadena de control de formato consta de especificadores de conversión, banderas, anchos de campos, precisiones y literales de carácter.
- Los enteros se imprimen con los siguientes especificadores de conversión: **`d`** o **`i`** para enteros con signo (opcional), **`o`** para enteros sin signo en forma octal, **`u`** para enteros sin signo en forma decimal, y **`x`** o **`X`** para enteros sin signo en forma hexadecimal. Los modificadores **`h`** o **`l`** son prefijos de los especificadores anteriores para indicar un entero corto o largo, respectivamente.
- Los valores de punto flotante se imprimen con los siguientes especificadores de conversión: **`e`** o **`E`** para la notación exponencial, **`f`** para la notación de punto flotante normal, **`g`** o **`G`** para la notación **`e`** (o **`E`**) o para la notación **`f`**. Cuando se indica el especificador de conversión **`g`** (o **`G`**), se utiliza el especificador de conversión **`e`** (o **`E`**) si el valor del exponente es menor que **`-4`**, o mayor o igual que la precisión con la que se imprime el valor.
- La precisión para los especificadores de conversión **`g`** y **`G`** indica el máximo número de dígitos significativos a imprimir.
- El especificador de conversión **`c`** imprime un carácter.
- El especificador de conversión **`s`** imprime una cadena de caracteres que termina con el carácter nulo.
- El especificador de conversión **`p`** despliega una dirección de una forma definida en la implementación (en muchos sistemas, utiliza la notación hexadecimal).
- El especificador de conversión **`n`** almacena el número de caracteres ya desplegados en la instrucción **`printf`**. El argumento correspondiente es un apuntador a un entero.
- El especificador de conversión **`%%`** provoca que se despliegue una literal **`%`**.

- Si el ancho del campo es mayor que la del objeto que se imprime, el objeto se justifica a la derecha de manera predeterminada.
- Los anchos de campo pueden utilizarse con todos los especificadores de conversión.
- La precisión que se utiliza con los especificadores de conversión indican el número mínimo de dígitos impresos. Si el valor contiene menos dígitos que la precisión especificada, en el valor a imprimir se colocan ceros como prefijos, hasta que el número de dígitos es equivalente a la precisión.
- La precisión utilizada con los especificadores de conversión de punto flotante **e**, **E** y **f** indican el número de dígitos que aparecen después del punto decimal. La precisión utilizada con los especificadores de conversión **g** y **G** indican el número de dígitos significativos que aparecerán.
- La precisión utilizada con el especificador de conversión **s** indica el número de caracteres a imprimir.
- La longitud y la precisión del campo se pueden combinar, colocando el ancho del campo seguido por un punto decimal, seguido por la precisión, entre el porcentaje y el especificador de conversión.
- Es posible especificar el ancho del campo y la precisión a través de expresiones enteras en la lista de argumentos que siguen a la cadena de control de formato. Para utilizar esta característica inserte un asterisco (*****), en lugar del ancho del campo o de la precisión. El argumento que coincide en la lista de argumentos se evalúa y se utiliza en lugar del asterisco. El valor del argumento puede ser negativo para el ancho del campo, pero debe ser positivo para la precisión.
- La bandera **–** justifica a la izquierda el argumento de un campo.
- La bandera **+** imprime un signo más para los valores positivos, y un signo menos para los valores negativos. La bandera espacio imprime un espacio que precede a un valor positivo, que no se despliega con la bandera **+**.
- La bandera **#** es prefijo de **0** para valores octales y **0x** o **0X** para valores hexadecimales, y fuerza la impresión del punto decimal para los valores de punto flotante impresos con **e**, **E**, **f**, **g** o **G** (por lo general, el punto decimal se despliega solamente si el valor contiene una parte fraccionaria).
- La bandera **0** imprime ceros al principio del campo para un valor que no ocupa completamente el ancho del campo.
- El formato preciso de entrada se lleva a cabo con la función **scanf** de la biblioteca.
- Los enteros se introducen con **scanf** mediante el especificador de conversión **d** e **i** para enteros con signo (opcional), y **o**, **u**, **x** o **X** para enteros sin signo. Los modificadores **h** y **l** se colocan antes del especificador de conversión para introducir un entero **short** o **long**, respectivamente.
- Los valores de punto flotante se introducen con **scanf** mediante el especificador de conversión **e**, **E**, **f**, **g** o **G**. Los modificadores **l** y **L** se colocan antes de cualquier especificador de conversión de punto flotante para indicar que el valor de entrada es un **double** o un **long double**, respectivamente.
- Los caracteres se introducen con **scanf** con el especificador de conversión **c**.
- Las cadenas se introducen con **scanf** con el especificador de conversión **s**.
- Un conjunto de exploración colocado en una **scanf** explora los caracteres de entrada, y busca solamente aquellos caracteres que coincidan con los caracteres contenidos en el conjunto de exploración. Cuando un carácter coincide, éste se almacena en el arreglo de caracteres. El conjunto de exploración detiene la entrada de caracteres cuando encuentra un carácter no contenido en el conjunto de exploración.
- Para crear un conjunto de exploración invertido, coloque un carácter tilde (**^**) dentro de los corchetes, antes de los caracteres de exploración. Esto provoca que los caracteres introducidos con **scanf** y que no aparecen en el conjunto de exploración se almacenen hasta que se encuentre un carácter contenido en el conjunto de exploración invertido.
- Los valores de direcciones se introducen con **scanf** mediante el especificador de conversión **p**.
- El especificador de conversión **n** almacena el número de caracteres previamente introducido por medio del **scanf** actual. El argumento correspondiente es un apuntador a **int**.
- El especificador de conversión **%%** con **scanf** hace coincidir un carácter **%** sencillo en la entrada.
- El carácter de supresión de asignación lee datos desde el flujo de entrada y descarta los datos.
- En **scanf**, un ancho de campo se utiliza para leer un número específico de caracteres desde el flujo de entrada.

## TERMINOLOGÍA

***** en la precisión
***** en una longitud del campo ancho de campo
**<stodio.h>**
alineación
bandera
bandera **-** (signo menos)
bandera **#**
bandera **+** (signo más)
bandera **0** (cero)
bandera espacio
cadena de control de formato
carácter de supresión de asignación **(*)**
circunflejo tilde (**^**)

conjunto de exploración
conjunto de exploración invertido
entero **long**
entero **short**
espacio en blanco
especificación de conversión
especificador de conversión **%**
especificador de conversión **c**
especificador de conversión **d**
especificador de conversión **e** o **E**
especificador de conversión **f**
especificador de conversión **g** o **G**
especificador de conversión **h**
especificador de conversión **i**
especificador de conversión **L**
especificador de conversión **l**
especificador de conversión **n**
especificador de conversión **o**
especificador de conversión **p**
especificador de conversión **s**
especificador de conversión **u**
especificador de conversión **x** (o **X**)
especificadores de conversión
especificadores enteros de conversión entera
flujo
flujo estándar de entrada
flujo estándar de error
flujo estándar de salida
formato de entero con signo
formato de entero sin signo
formato exponencial de punto flotante
formato hexadecimal
formato octal
inserción de espacio
inserción de un carácter de impresión
justificación a la derecha
justificación a la izquierda
literales de carácter
longitud ancho del campo
notación científica
precisión
**printf**
punto flotante
redirección de un flujo
redondeo
**scanf**
secuencia de escape
secuencia de escape **\?**
secuencia de escape ****
secuencia de escape **\'**
secuencia de escape **\"**
secuencia de escape **\a**
secuencia de escape **\b**
secuencia de escape **\f**
secuencia de escape **\n**
secuencia de escape **\r**
secuencia de escape **\t**
secuencia de escape **\v**

## ERRORES COMUNES DE PROGRAMACIÓN

**9.1** Olvidar encerrar una cadena de control de formato entre comillas, es un error de sintaxis.

**9.2** Imprimir un valor negativo con un especificador de conversión que espera un valor **unsigned**.

**9.3** Utilizar un **%c** para imprimir una cadena es un error. El especificador de conversión **%c** espera un **char** como argumento. Una cadena es un apuntador a **char** (es decir, un **char ***).

**9.4** En algunos sistemas, utilizar un **%s** para imprimir un argumento **char**, provoca un error fatal en tiempo de ejecución llamado violación de acceso. El especificador de conversión **%s** espera un argumento de tipo apuntador a **char**.

**9.5** Utilizar comillas sencillas alrededor de cadenas de caracteres es un error de sintaxis. Las cadenas de caracteres deben encerrarse entre comillas dobles.

**9.6** Utilizar comillas dobles alrededor de una constante de carácter crea una cadena que consiste en dos caracteres, en la cual el segundo carácter es el nulo de terminación. Una constante de carácter es un carácter individual encerrado entre comillas sencillas.

**9.7** Intentar imprimir una literal del carácter de porcentaje mediante el uso de **%** en lugar de **%%** dentro de la cadena de control de formato, es un error. Cuando aparece **%** en una cadena de control de formato, debe ser seguida por un especificador de conversión.

**9.8** No proporcionar un ancho de campo suficiente para manipular un valor de impresión puede ocasionar el desplazamiento otros datos en la impresión y producir salidas confusas. ¡Conozca sus datos!

**9.9** Intentar imprimir una comilla sencilla, una comilla doble, un signo de interrogación o una diagonal invertida como un dato literal dentro de una instrucción **printf** sin colocar una diagonal invertida para formar una secuencia de escape, es un error de sintaxis.

## TIP PARA PREVENIR ERRORES

**9.1** Cuando imprima datos, asegúrese de que el usuario sea consciente de las situaciones en las que los datos pudieran ser imprecisos debido al formato (por ejemplo, errores de redondeo debido a las especificaciones de la precisión).

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**9.1** Por presentación, edite de manera clara las salidas de un programa, para hacer que éstas sean más legibles y para reducir los errores de usuario.

**9.2** Cuando se introduzcan datos, solicite al usuario uno o varios elementos a la vez. Evite pedir al usuario que introduzca muchos elementos en respuesta a una sola indicación.

## TIP DE PORTABILIDAD

**9.1** El especificador de conversión `p` despliega una dirección de manera definida en la implementación (en muchos sistemas, se utiliza la notación hexadecimal en lugar de la notación decimal).

## EJERCICIOS DE AUTOEVALUACIÓN

**9.1** Complete los espacios en blanco:

a) Toda la entrada y la salida de datos se lleva a cabo en forma de ____________.
b) Por lo general, el flujo de ____________ está conectado al teclado.
c) Por lo general, el flujo de ____________ está conectado a la pantalla de la computadora.
d) El formato preciso de salida se logra con la función ____________.
e) La cadena de control de formato puede contener ____________, ____________, ____________ y ____________.
f) Se pueden utilizar los especificadores de conversión ____________ o ____________ para mostrar un entero decimal con signo.
g) Los especificadores de conversión ____________, ____________ o ____________ se utilizan para mostrar enteros sin signo en forma octal, decimal, hexadecimal, respectivamente.
h) Los modificadores ____________ y ____________ se colocan antes del especificador de conversión de enteros para indicar que se desplegarán valores `short` o `long`.
i) Los especificadores de conversión ____________ y ____________ se utilizan para desplegar valores de punto flotante en notación exponencial.
j) El modificador ____________ se coloca antes de cualquier especificador de conversión de punto flotante para indicar que se desplegará un valor `long double`.
k) Los especificadores de conversión `e`, `E` y `f` se despliegan con ____________ dígitos de precisión a la derecha del punto decimal, si no se especifica la precisión.
l) Los especificadores de conversión ____________ y ____________ se utilizan para imprimir cadenas y caracteres, respectivamente.
m) Todas las cadenas terminan con el carácter ____________.
n) El ancho del campo y la precisión en el especificador de conversión de `printf` pueden controlarse con expresiones enteras, sustituyendo con un ____________ el ancho del campo o la precisión y colocando una expresión entera en el argumento correspondiente de la lista de argumentos.
o) La bandera ____________ provoca la justificación izquierda de la salida dentro de un campo.
p) La bandera ____________ provoca que las variables se desplieguen con un signo más o un signo menos.
q) El formato preciso de entrada se consigue con la función ____________.
r) Un ____________ se utiliza para explorar una cadena, en busca de caracteres específicos y para almacenarlos dentro de un arreglo.
s) El especificador de conversión ____________ puede utilizarse para introducir enteros octales, decimales y hexadecimales con signo (opcional).
t) El especificador de conversión ____________ se puede utilizar para introducir un valor `double`.
u) La ____________ se utiliza para leer datos desde el flujo de entrada y descartarlos sin asignarlos a una variable.
v) Un ____________ se puede utilizar en un especificador de conversión de `scanf` para indicar que se debe leer un número específico de caracteres o dígitos desde el flujo de entrada.

**9.2** Encuentre el error en cada una de las siguientes instrucciones, y explique cómo puede corregirlo.

a) La siguiente instrucción debe imprimir el carácter `'c'`.
```
printf( "%s\n", 'c' );
```
b) La siguiente instrucción debe imprimir `9.375%`.
```
printf( "%.3f%", 9.375 );
```
c) La siguiente instrucción debe imprimir el primer carácter de la cadena `"Lunes"`.
```
printf( "%c\n", "Lunes" );
```
d) `printf( ""Una cadena entre comillas"");`
e) `printf( %d%d, 12, 20 );`
f) `printf( "%c", "x" );`
g) `printf( "%s\n", 'Ricardo' );`

**9.3** Escriba una instrucción para cada una de las siguientes tareas:

a) Imprima `1234` justificado a la izquierda en un campo de `10` dígitos.

b) Imprima **123.456789** en notación exponencial con signo (**+** o **-**) y tres dígitos de precisión.
c) Lea un valor **double** dentro de la variable **numero**.
d) Imprima **100** en forma octal, precedido por **0**.
e) Lea una cadena dentro del arreglo de caracteres **cadena**.
f) Lea los caracteres del arreglo **n** hasta que encuentre un carácter que no sea un dígito.
g) Utilice las variables enteras **x** y **y** para especificar el ancho del campo y la precisión utilizada para desplegar el valor **double 87.4573**.
h) Lea un **valor** de la forma **3.5%.** Almacene el porcentaje en una variable **float** llamada **porcentaje**, y elimine el **%** del flujo de entrada. No utilice el carácter de supresión de asignación.
i) Imprima **3.333333** como un valor long **double** con un signo (**+** o **-**), en un campo de **20** caracteres con una precisión de **3**.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**9.1** a) Flujos. b) Entrada estándar. c) Salida estándar. d) **printf**. e) Especificadores de conversión, banderas, anchos de campo, precisiones, literales de carácter. f) **d**, **i**. g) **o**, **u**, **x** (o **X**). h) **h**, **l**. i) **e** (o **E**). j) **L**. k) **6**. l) **s**, **c**. m) **NULL** (**'\0'**). n) asterisco (*****). o) **-** (menos). p) **+** (más). q) **scanf**. r) Conjunto de exploración. s) **i**. t) **le**, **lE**, **lf**, **lg** o **lG**. u) Carácter de supresión de asignación (*****). v) Ancho del campo.

**9.2** a) Error: el especificador de conversión **s** espera un argumento de tipo apuntador a **char**.
Corrección: para imprimir el carácter **'c'**, utilice el especificador de conversión **%c**, o cambie de **'c'** a **"c"**.
b) Error: intentar imprimir la litera de carácter **%** sin utilizar el especificador de conversión **%%**.
Corrección: utilice **%%** para imprimir la literal de carácter **%**.
c) Error: el especificador de conversión **c** espera un argumento de tipo **char**.
Corrección: para imprimir el primer carácter de **"Lunes"**, utilice el especificador de conversión **%1s**.
d) Error: tratar de imprimir la literal de carácter **"** sin la secuencia de escape **\"**.
Corrección: reemplace cada comilla del conjunto interno de comillas con **\"**.
e) Error: la cadena de control de formato no se encuentra encerrada entre comillas dobles.
Corrección: encierre **%d%d** entre comillas dobles.
f) Error: el carácter **x** está encerrado entre comillas dobles.
Corrección: las constantes de carácter que se imprimen con **%c** se deben encerrar entre comillas sencillas.
g) Error: la cadena a imprimir está encerrada entre comillas sencillas.
Corrección: utilice comillas dobles en lugar de comillas sencillas para representar una cadena.

**9.3** a) `printf( "%10d\n", 1234 );`
b) `printf( "%+.3e\n", 123.456789 );`
c) `printf( "%lf", &numero );`
d) `printf( "%#o\n", 100 );`
e) `scanf( "%s", cadena );`
f) `scanf( "%[0123456789]", n );`
g) `printf( "%*.*f\n", x, y, 87.4573 );`
h) `scanf( "%f%%", &porcentaje );`
i) `printf( "%+20.3Lf\n", 3.333333 );`

## EJERCICIOS

**9.4** Escriba una instrucción **printf** o **scanf** para cada una de las siguientes tareas:
a) Imprima el entero sin signo **40000** justificado a la izquierda, dentro de un campo de **15** posiciones con **8** dígitos.
b) Lea un valor hexadecimal dentro de la variable **hex**.
c) Imprima **200** con y sin signo.
d) Imprima **100** en forma hexadecimal, precedido por **0x**.
e) Lea los caracteres dentro del arreglo **s**, hasta que encuentre la letra **p**.
f) Imprima **1.234** en un campo de **9** posiciones, precedido por ceros.
g) Lea la hora de la forma **hh:mm:ss**; almacene las partes de la hora en las variables enteras **hora**, **minuto** y **segundo**. Ignore los dos puntos (**:**) del flujo de entrada. Utilice el carácter de supresión de asignación.
h) Lea una cadena de la forma **"caracteres"** desde la entrada estándar. Almacene la cadena dentro del arreglo caracteres **s**. Elimine las comillas del flujo de entrada.

i) Lea la hora de la forma **hh:mm:ss**; almacene las partes de la hora en las variables enteras **hora**, **minuto** y **segundo**. Ignore los dos puntos (**:**) del flujo de entrada. No utilice el carácter de supresión de asignación.

**9.5** Muestre lo que imprime cada una de las siguientes instrucciones. Si la instrucción es incorrecta, indique por qué.

a) **printf( "%-10d\n", 10000 );**
b) **printf( "%c\n", "Esta es una cadena" );**
c) **printf( "%*.*lf\n", 8, 3, 1024.987654 );**
d) **printf( "%#o\n%#X\n%#e\n", 17, 17, 1008.83689 );**
e) **printf( "% ld\n%+ld\n", 1000000, 1000000 );**
f) **printf( "%10.2E\n", 444.93738 );**
g) **printf( "%10.2g\n", 444.93738 );**
h) **printf( "%d\n", 10.987 );**

**9.6** Encuentre el error en cada uno de los siguientes segmentos de programa. Explique cómo se puede corregir cada error.

a) **printf( "%s\n", 'Feliz Cumpleanios' );**
b) **printf( "%c\n", 'Hola' );**
c) **printf( "%c\n", "Esta es una cadena" );**
d) La siguiente instrucción debe imprimir **"Buen Viaje"**:
**printf( ""%s"", "Buen Viaje" );**
e) **char dia[] = "Domingo";**
**printf( "%s\n", dia[ 3 ] );**
f) **printf( 'Introduzca su nombre: ' );**
g) **printf( %f, 123.456 );**
h) La siguiente instrucción debe imprimir los caracteres **'O'** y **'K'**:
**printf( "%s%s\n", 'O', 'K' );**
i) **char s[ 10 ];**
**scanf( "%c", s[ 7 ] );**

**9.7** Escriba un programa que cargue un arreglo de 10 elementos llamado **numero**, que lea enteros al azar entre 1 y 1000. Por cada valor, imprima el valor y un total del número de caracteres impresos. Utilice el especificador de conversión **%n** para determinar el número de caracteres de salida para cada valor. Imprima el número total de caracteres de salida para todos los valores cargados, incluso el valor actual cada vez que se imprima. La salida del programa debe tener el siguiente formato:

```
Valor        Caracteres totales
342          3
1000         7
963          10
6            11
etc.
```

**9.8** Escriba un programa que evalúe la diferencia entre los especificadores de formato **%d** y **%i**, cuando se utilizan en la instrucciones

```
scanf( "%i%d", &x, &y );
printf( "%d %d\n", x, y );
```

para imprimir los valores de salida. Pruebe el programa con el siguiente conjunto de datos de entrada.

```
  10      10
 -10     -10
 010     010
0x10    0x10
```

**9.9** Escriba un programa que imprima un apuntador por medio de los especificadores de conversión entera y de conversión **%p**. ¿Cuál de éstos imprime valores extraños? ¿Cuál provoca errores? ¿En qué formato el especificador de conversión **%p** despliega la dirección en su sistema?

**9.10** Escriba un programa que evalúe los resultados de la impresión del valor entero **12345** y del valor de punto flotante **1.2345** en distintos tamaños de campo. ¿Qué sucede cuando los valores se imprimen dentro de campos que contienen menos dígitos que valores?

**9.11** Escriba un programa que imprima el valor **100.453627** redondeado al dígito más cercano, décima, centésima, milésima y diezmilésima.

**9.12** Escriba un programa que imprima una cadena desde el teclado y que determine la longitud de la cadena. Imprima la cadena usando el doble de la longitud de la cadena como el ancho de campo.

**9.13** Escriba un programa que convierta las temperaturas enteras de **0** a **212** grados Fahrenheit a temperaturas en grados Celsius expresadas en punto flotante con **3** dígitos de precisión. Utilice la fórmula

```
celsius = 5.0 / 9.0 * ( fahrenheit – 32 );
```

para realizar el cálculo. La salida debe imprimirse en columnas de dos dígitos justificadas a la derecha con 10 caracteres cada una, y las temperaturas Celsius deben ser precedidas por un signo, tanto para valores positivos como para negativos.

**9.14** Escriba un programa que pruebe todas las secuencias de escape de la figura 9.16. Para las secuencias de escape que mueven el cursor, imprima el carácter antes y después de imprimir la secuencia de escape, de tal modo que sea claro hacia dónde se mueve el cursor.

**9.15** Escriba un programa que determine en dónde puede imprimirse el carácter **?** como parte de una cadena de control de formato de **printf**, como una literal de carácter en lugar de una secuencia de escape **\?**.

**9.16** Escriba un programa que introduzca el valor **437** con cada uno de los especificadores de conversión de **scanf**. Imprima cada valor de entrada con todos los especificadores de conversión entera.

**9.17** Escriba un programa que utilice cada uno de los especificadores de conversión **e**, **f** y **g** para introducir el valor **1.2345**. Imprima los valores de cada variable para probar que puede utilizarse cada especificador de conversión para introducir el mismo valor.

**9.18** En algunos lenguajes de programación, las cadenas se introducen encerradas entre comillas sencillas o dobles. Escriba un programa que lea las tres cadenas **suzy**, **“suzy”** y **‘suzy’**. ¿C ignora las comillas sencillas y dobles, o las lee como parte de la cadena?

**9.19** Escriba un programa que determine en dónde puede imprimirse el carácter **?** como una constante de carácter **‘?’**, en lugar de una constante de secuencia de escape **‘\?’** con el especificador de conversión **%c** dentro de la cadena de control de formato de una instrucción **printf**.

**9.20** Escriba un programa que utilice el especificador de conversión **g** para imprimir el valor **9876.12345**. Imprima el valor con precisiones en el rango de **1** a **9**.

# 10

# Estructuras, uniones, manipulaciones de bits y enumeraciones en C

## Objetivos

- Crear y utilizar estructuras, uniones y enumeraciones.
- Pasar estructuras a funciones por valor y por referencia.
- Manipular datos con los operadores a nivel de bits.
- Crear campos de bits para almacenar datos de manera compacta.




*Nunca pude entender lo que esos malditos puntos significaban.*
Winston Churchill

*Incluso unidos en la separación.*
William Shakespeare

*Puedes incluirme.*
Samuel Goldwyn

*La misma vieja y piadosa mentira*
*se repite conforme pasa el tiempo*
*y permanentemente implica un golpe*
*"¡Realmente no has cambiado nada!"*
Margaret Fishback

## Plan general



## 10.1 Introducción

Las *estructuras*, en ocasiones conocidas como *agregados*, son colecciones de variables relacionadas bajo un nombre. Las estructuras pueden contener variables de diferentes tipos de datos, a diferencia de los arreglos, los cuales sólo contienen elementos del mismo tipo. Las estructuras generalmente se utilizan para definir registros que van a almacenarse en archivos (vea el capítulo 11). Los apuntadores y las estructuras facilitan la formación de estructuras de datos más complejas, como listas ligadas, colas, pilas y árboles (vea el capítulo 12).

## 10.2 Definición de estructuras

Las estructuras son *tipos de datos derivados*, que se construyen por medio de objetos de otros tipos. Considere la siguiente definición de una estructura:

```
struct carta {
   char *cara;
   char *palo;
};
```

La palabra reservada **`struct`** introduce la definición de una estructura. El identificador **`carta`** es la *etiqueta de la estructura*, la cual da nombre a la definición de la estructura y se utiliza con la palabra reservada **`struct`** para declarar variables de *tipo estructura*. En este ejemplo, el tipo estructura es **`struct carta`**. Las variables declaradas dentro de las llaves de la definición de una estructura son *miembros de la estructura*. Los miembros de una misma estructura deben tener nombres únicos, pero dos estructuras diferentes pueden contener miembros con el mismo nombre, sin problema (pronto veremos por qué). Toda definición de una estructura debe terminar con un punto y coma.

**Error común de programación 10.1**

*Olvidar el punto y coma al finalizar la definición de una estructura, es un error de sintaxis.*

La definición de **`struct carta`** contiene dos miembros de tipo **`char*`**: **`cara`** y **`palo`**. Los miembros de una estructura pueden ser variables de tipos de datos primitivos (por ejemplo, **`int`**, **`float`**, etcétera), o agregados, como arreglos y otras estructuras. Como vimos en el capítulo 6, cada elemento de un arreglo debe ser del mismo tipo. Sin embargo, los miembros de una estructura pueden ser de diversos tipos. Por ejemplo,

```
struct empleado {
   char nombre[ 20 ];
```

```
      char apellido[ 20 ];
      int edad;
      char sexo;
      double salarioPorHora;
   };
```

contiene miembros que son arreglos de caracteres para el nombre y el apellido, y un miembro **int** para la edad del empleado, un miembro **char** que puede contener **'M'** o **'F'** para el sexo del empleado, y un miembro **double** para el salario por hora del empleado.

Una estructura no puede contener una instancia de sí misma. Por ejemplo, una variable de tipo **struct** empleado no puede declararse en la definición de **struct empleado**. Sin embargo, un apuntador a **struct empleado** puede incluirse. Por ejemplo,

```
   struct empleado2 {
      char nombre[ 20 ];
      char apellido[ 20 ];
      int edad;
      char sexo;
      double salarioPorHora;
      struct empleado2 persona; /* ERROR */
      struct empleado2 *ptrE; /* apuntador */
   };
```

**struct empleado2** contiene una instancia de sí misma (**persona**), lo cual es un error. Debido a que **ptrE** es un apuntador (al tipo **struct empleado2**), si es permitido en la definición. A una estructura que contiene un miembro que es un apuntador a la misma estructura se le conoce como estructura autorreferenciada. Las estructuras autorreferenciadas se utilizan en el capítulo 12 para construir estructuras de datos ligadas.

Las definiciones de estructuras no reservan espacio alguno en memoria; en cambio, cada definición crea un nuevo tipo de dato que se utiliza para definir variables. Las variables de tipo estructura se definen como las variables de otros tipos. La definición

```
   struct carta unaCarta, mazo[ 52 ], *ptrCarta;
```

declara **unaCarta** para que sea una variable del tipo **struct carta**; también declara **mazo** para que sea un arreglo del tipo **struct carta** de 52 elementos y declara **ptrCarta** para que sea un apuntador a **struct carta**. Las variables de un tipo de estructura dado también pueden declararse colocando una lista separada por comas con los nombres de las variables entre la llave que cierra la definición de la estructura y el punto y coma que finaliza la definición de la estructura. Por ejemplo, la definición anterior pudo haberse incorporado en la definición de la estructura **struct carta**, de la siguiente forma:

```
   struct carta {
      char *cara;
      char *palo;
   } unaCarta, mazo[ 52 ], *ptrCarta;
```

La etiqueta con el nombre de la estructura es opcional. Si la definición de una estructura no contiene una etiqueta con su nombre, las variables del tipo estructura pueden declararse solamente en la definición de la estructura y no en una declaración separada.



### Buena práctica de programación 10.1

*Cuando genere un tipo de estructura, siempre proporcione una etiqueta con su nombre. Dicha etiqueta es conveniente para que posteriormente se declaren nuevas variables correspondientes a la estructura.*



### Buena práctica de programación 10.2

*Elegir una etiqueta con un nombre significativo ayuda a que un programa esté autodocumentado.*

Las únicas operaciones válidas que pueden realizarse con estructuras son las siguientes: asignación de variables de estructuras a variables de estructuras del mismo tipo, tomar la dirección (**&**) de una variable de estructura, acceder a los miembros de una variable de estructura (vea la sección 10.4) y utilizar el operador **sizeof** para determinar el tamaño de una variable de estructura.



**Error común de programación 10.2**

*Asignar una estructura de un tipo a una estructura de diferente tipo, es un error de compilación.*

Las estructuras no pueden compararse por medio de operadores **==**y **!=**, ya que los miembros de estructuras no necesariamente se almacenan en bytes de memoria consecutivos. Algunas veces existen "huecos" en una estructura, debido a que las computadoras pueden almacenar tipos de datos específicos sólo en ciertos límites de memoria, como los de media, una o dos palabras. Una palabra es la unidad estándar de memoria utilizada para almacenar datos en una computadora; por lo general, 2 o 4 bytes. Considere la siguiente definición de una estructura, en la que se declaran **muestra1** y **muestra2** del tipo **struct** ejemplo:

```
struct ejemplo {
   char c;
   int i;
} muestra1, muestra2;
```

Una computadora con palabras de 2 bytes podría necesitar que cada miembro de **struct ejemplo** estuviera alineado de acuerdo con un límite de palabras, es decir, al principio de una palabra (esto depende de cada máquina). La figura 10.1 muestra un ejemplo de la alineación de almacenamiento para una variable del tipo **struct ejemplo**, a la que se le ha asignado el carácter **'a'** y el entero **97** (la representación en bits de los valores que muestran). Si los miembros se almacenan al comienzo de los límites de palabras, hay un hueco de 1 byte (el byte **1** de la figura) en el almacenamiento de las variables de **struct ejemplo**. El valor en el hueco de 1 byte es indefinido. Incluso si los valores de los miembros **muestra1** y **muestra2** son iguales, las estructuras no necesariamente son iguales, ya que es muy poco probable que los huecos indefinidos de 1 byte contengan valores idénticos.



**Error común de programación 10.3**

*Comparar estructuras es un error de sintaxis.*



**Tip de portabilidad 10.1**

*Debido a que el tamaño de los elementos de un tipo en particular depende de la máquina, y debido a que las consideraciones de alineación de almacenamiento también dependen de la máquina, la representación de una estructura también depende de la máquina.*

## 10.3 Inicialización de estructuras

Las estructuras pueden inicializarse por medio de listas de inicialización, como en los arreglos. Para inicializar una estructura, coloque un signo de igualdad después del nombre de la variable y, entre llaves, una lista de inicializadores separada por comas. Por ejemplo, la declaración

```
struct carta unaCarta = { "Tres", "Corazones" };
```

crea una variable **unaCarta** para que sea del tipo **struct carta** (como la definimos en la sección 10.2) e inicializa el miembro **cara** en **"Tres"** y el miembro **palo** en **"Corazones"**. Si en una lista existen menos inicializadores que miembros de la estructura, los miembros restantes se inicializan automáticamente en **0** (o en **NULL**, si el miembro es un apuntador). Las variables de estructuras definidas fuera de la definición de una función (es decir, externamente) se inicializan en **0** o **NULL**, si no se inicializan explícitamente en la definición externa. Las variables de estructuras también pueden inicializarse en instrucciones de asignación, asignando una variable de estructura del mismo tipo, o asignando valores a los miembros individuales de la estructura.

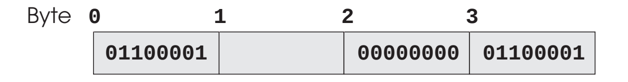



**Figura 10.1** Posible alineación de almacenamiento para una variable del tipo **struct ejemplo**, la cual muestra un área indefinida de memoria.

## 10.4 Acceso a miembros de estructuras

Se utilizan dos operadores para acceder a los miembros de estructuras: el *operador miembro de la estructura* (**.**), también llamado *operador punto*, y el *operador apuntador de la estructura* (**->**), también llamado *operador flecha*. El operador miembro de la estructura accede al miembro de la estructura a través del nombre de ésta. Por ejemplo, para imprimir el miembro **palo** de la variable **unaCarta**, que definimos en la sección 10.3, utilizamos la instrucción

```
printf( “%s”, unaCarta.palo ); /* despliega Corazones */
```

El operador apuntador de la estructura, que consiste en un signo de menos (**-**) y uno de mayor que (**>**) sin espacios que los separen, accede al miembro de la estructura a través de un *apuntador a la estructura*. Suponga que el apuntador **ptrCarta** se declaró para que apuntara a **struct carta**, y que la dirección de la estructura **unaCarta** se asignó a **ptrCarta**. Para imprimir el miembro **palo** de la estructura **unaCarta** con el apuntador **ptrCarta**, utilice la instrucción

```
printf( “%s”, ptrCarta->palo ); /* despliega Corazones */
```

La expresión **ptrCarta->palo** es equivalente a **(*ptrCarta).palo**, la cual desreferencia al apuntador y accede al miembro palo por medio del operador miembro de la estructura. Aquí, los paréntesis son necesarios porque el operador miembro de la estructura (**.**) tiene una precedencia más alta que el operador de desreferencia del apuntador (*****). El operador apuntador de la estructura y el operador miembro de la estructura, junto con los paréntesis (para llamar funciones) y los corchetes (**[]**) utilizados para colocar subíndices a los arreglos, tienen la precedencia de operadores más alta y asocian de izquierda a derecha.



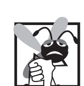
**Tip para prevenir errores 10.1**

*Evite utilizar los mismos nombres para los miembros de estructuras de diferentes tipos. Esto está permitido, sin embargo, puede ocasionar confusión.*



**Buena práctica de programación 10.3**

*No coloque espacios alrededor de los operadores (**->**) y (**.**). Omitir los espacios ayuda a enfatizar que las expresiones en las que están contenidos los operadores son esencialmente nombres de variables.*



**Error común de programación 10.4**

*Insertar un espacio entre los componentes **–** y **>** del operador apuntador de la estructura (o insertar espacios entre los componentes de cualquier otro operador con combinación de teclas, excepto **?:**), es un error de sintaxis.*



**Error común de programación 10.5**

*Intentar hacer referencia a un miembro de una estructura utilizando únicamente el nombre del miembro, es un error de sintaxis.*



**Error común de programación 10.6**

*No utilizar paréntesis cuando se hace referencia al miembro de una estructura que utiliza un apuntador y el operador miembro de la estructura (por ejemplo, ***ptrCarta.palo**), es un error de sintaxis.*

El programa de la figura 10.2 muestra el uso de los operadores miembro y apuntador de la estructura. Por medio del operador miembro de la estructura, a los miembros de la estructura **unaCarta** se les asignan los valores **“As”** y **“Espadas”**, respectivamente (líneas 18 y 19). Al apuntador **ptrCarta** se le asigna la dirección de la estructura **unaCarta** (línea 21). La función **printf** imprime los miembros de la estructura **unaCarta** por medio del operador miembro de la estructura con el nombre de la variable **unaCarta**, el operador apuntador de la estructura con el apuntador **ptrCarta** y el operador miembro de la estructura con el apuntador desreferenciado **ptrCarta** (líneas 23 a 25).

```
1  /* Figura 10.2: fig10_02.c
2     Uso de los operadores de estructura
```

**Figura 10.2** Operador miembro de la estructura y operador apuntador de la estructura. (Parte 1 de 2.)

```
   miembro y de apuntador a estructura */
#include <stdio.h>

/* definición de la estructura carta */
struct carta {
   char *cara; /* define el apuntador cara */
   char *palo; /* define el apuntador palo */
}; /* fin de la estructura carta */

int main()
{
   struct carta unaCarta;  /* define una estructura variable carta */
   struct carta *ptrCarta; /* define un apuntador a una estructura carta */

   /* coloca cadenas dentro de unaCarta */
   unaCarta.cara = “As”;
   unaCarta.palo = “Espadas”;

   ptrCarta = &unaCarta; /* asigna la dirección de unaCarta a ptrCarta */

   printf( “%s%s%s\n%s%s%s\n%s%s%s\n”, unaCarta.cara, “ de “, unaCarta.palo,
      ptrCarta->cara, “ de “, ptrCarta->palo,
      ( *ptrCarta ).cara, “ de “, ( *ptrCarta ).palo );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
As de Espadas
As de Espadas
As de Espadas
```

**Figura 10.2** Operador miembro de la estructura y operador apuntador de la estructura. (Parte 2 de 2.)

## 10.5 Uso de estructuras con funciones

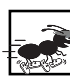
Las estructuras pueden pasarse a funciones, pasando miembros individuales de la estructura, pasando una estructura completa o pasando un apuntador a una estructura. Cuando las estructuras o los miembros individuales de la estructura pasan a una función, pasan por valor. Por lo tanto, los miembros de una estructura que llama no pueden ser modificados por la función llamada.

Para pasar una estructura por referencia, pase la dirección de la variable estructura. Los arreglos de estructuras, como todos los demás arreglos, se pasan automáticamente por referencia.

En el capítulo 6, dijimos que un arreglo podía pasarse por valor, utilizando una estructura. Para pasar por valor a un arreglo, genere una estructura con el arreglo como un miembro. Las estructuras se pasan por valor, por lo que el arreglo se pasa por valor.

**Error común de programación 10.7**

*Suponer que las estructuras, al igual que los arreglos, se pasan automáticamente por referencia, e intentar modificar los valores de la estructura que llama en la función llamada, es un error lógico.*



**Tip de rendimiento 10.1**

*Pasar estructuras por referencia resulta más eficiente que pasarlas por valor (lo cual requiere que se copie la estructura completa).*

## 10.6 typedef

La palabra reservada **typedef** proporciona un mecanismo para crear sinónimos (o alias) de tipos de datos definidos previamente. Los nombres de los tipos de estructuras con frecuencia se definen con **typedef**, para crear nombres cortos. Por ejemplo, la instrucción

```
typedef struct carta Carta;
```

define el nuevo nombre de tipo **Carta** como un sinónimo del tipo **struct carta**. Los programadores en C con frecuencia utilizan **typedef** para definir tipos de estructuras, por lo que no se necesita una etiqueta para la estructura. Por ejemplo, la siguiente definición

```
typedef struct {
   char *cara;
   char *palo;
} Carta;
```

crea la estructura **Carta** sin la necesidad de una instrucción aparte **typedef**.



### Buena práctica de programación 10.4

*Escriba en mayúscula la primera letra de los nombres de **typedef** para enfatizar que esos nombres son sinónimos de los otros nombres de tipos.*

Ahora, **Carta** puede utilizarse para declarar variables del tipo **struct carta**. La declaración

```
Carta mazo[ 52 ];
```

declara un arreglo de 52 estructuras **Carta** (es decir, variables del tipo **struct carta**). Al crear un nuevo nombre con **typedef**, no se genera un tipo nuevo; **typedef** simplemente crea un nuevo nombre de tipo, el cual puede usarse como un alias de un nombre existente. Un nombre significativo ayuda a autodocumentar el programa. Por ejemplo, cuando leemos la declaración anterior sabemos que "**mazo** es un arreglo de 52 **Cartas**".

Con frecuencia, **typedef** se utiliza para crear sinónimos de los tipos de datos básicos. Por ejemplo, un programa que requiere enteros de 4 bytes puede utilizar el tipo **int** en un sistema, y el tipo **long** en otro. Los programas diseñados para su portabilidad, con frecuencia utilizan **typedef** para crear un alias para enteros de 4 bytes, como **Entero**. El alias **Entero** puede modificarse una vez en el programa para hacer que éste funcione en ambos sistemas.



### Tip de portabilidad 10.2

*Utilice **typedef** para ayudar a que un programa sea más portable.*

## 10.7 Ejemplo: Simulación de alto rendimiento para barajar y repartir cartas

El programa de la figura 10.3 se basa en el programa para barajar y repartir cartas que explicamos en el capítulo 7. El programa representa el mazo de cartas como un arreglo de estructuras. El programa utiliza algoritmos de alto rendimiento para barajar y repartir. La salida de este programa aparece en la figura 10.4.

```
/* Figura 10.3: fig10_03.c
   Programa para barajar y repartir con el uso de estructuras */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* definición de la estructura carta */
struct carta {
   const char *cara; /* define el apuntador cara */
   const char *palo; /* define el apuntador palo */
}; /* fin de la estructura carta */
```

**Figura 10.3** Simulación de alto rendimiento para barajar y repartir cartas. (Parte 1 de 3.)

```

typedef struct carta Carta; /* nuevo tipo de nombre para la estructura
                               baraja */

/* prototipos */
void llenaMazo( Carta * const wMazo, const char * wCara[],
   const char * wPalo[] );
void barajar( Carta * const wMazo );
void repartir( const Carta * const wMazo );

int main()
{
   Carta mazo[ 52 ]; /* define el arreglo Carta */

   /* inicializa el arreglo de apuntadores */
   const char *cara[] = { “As”, “Dos”, “Tres”, “Cuatro”, “Cinco”,
      “Seis”, “Siete”, “Ocho”, “Nueve”, “Diez”,
      “Joto”, “Quina”, “Rey”};

   /* inicializa el arreglo de apuntadores */
   const char *palo[] = { “Corazones”, “Diamantes”, “Treboles”, “Espadas”};

   srand( time( NULL ) ); /* randomizar */

   llenaMazo( mazo, cara, palo ); /* carga el mazo con las barajas */
   barajar( mazo );  /* coloca la Baraja en orden aleatorio */
   repartir( mazo ); /* reparte las 52 barajas */

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* coloca cadenas dentro de las estructuras Baraja */
void llenaMazo( Carta * const wMazo, const char * wCara[],
   const char * wPalo[] )
{
   int i; /* contador */

   /* ciclo a través de wMazo */
   for ( i = 0; i <= 51; i++ ) {
      wMazo[ i ].cara = wCara[ i % 13 ];
      wMazo[ i ].palo = wPalo[ i / 13 ];
   } /* fin de for */

} /* fin de la función llenaMazo */

/* baraja el mazo */
void barajar( Carta * const wMazo )
{
   int i;     /* contador */
   int j;     /* variable para almacenar el valor aleatorio entre 0 - 51 */
   Carta temp; /* define la estructura temporal para intercambiar cartas */

   /* ciclo a través de wMazo para intercambiar aleatoriamente Baraja */
   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wMazo[ i ];
```

**Figura 10.3** Simulación de alto rendimiento para barajar y repartir cartas. (Parte 2 de 3.)

```
68          wMazo[ i ] = wMazo[ j ];
69          wMazo[ j ] = temp;
70       } /* fin de for */
71
72    } /* fin de la función barajar */
73
74    /* reparte la baraja */
75    void repartir( const Carta * const wMazo )
76    {
77       int i; /* contador */
78
79       /* ciclo a través de wMazo */
80       for ( i = 0; i <= 51; i++ ) {
81          printf( "%5s de %-8s%c", wMazo[ i ].cara, wMazo[ i ].palo,
82             ( i + 1 ) % 2 ? '\t' : '\n' );
83       } /* fin de for */
84
85    } /* fin de la función repartir */
```

**Figura 10.3** Simulación de alto rendimiento para barajar y repartir cartas. (Parte 3 de 3.)

```
Tres de Diamantes        Ocho de Espadas
Siete de Corazones       Nueve de Diamantes
 Joto de Espadas           Rey de Treboles
Cuatro de Treboles        Diez de Corazones
  Rey de Diamantes        Joto de Treboles
 Joto de Corazones        Seis de Diamantes
Nueve de Corazones       Cuatro de Diamante
 Ocho de Diamantes       Quina de Diamantes
 Ocho de Treboles         Diez de Diamantes
   As de Diamantes          As de Espadas
Cinco de Treboles        Nueve de Espadas
 Diez de Treboles          Rey de Espadas
  Rey de Corazones        Seis de Treboles
  Dos de Diamantes          As de Treboles
   As de Corazones         Dos de Espadas
Cuatro de Espadas        Nueve de Treboles
Quina de Corazones       Cinco de Diamantes
 Tres de Treboles        Siete de Treboles
Siete de Espadas         Cinco de Corazones
 Joto de Diamantes       Cinco de Espadas
Quina de Treboles         Ocho de Corazones
  Dos de Treboles         Tres de Espadas
 Seis de Espadas          Seis de Corazones
 Tres de Corazones       Siete de Diamantes
```

**Figura 10.4** Salida de la simulación de alto rendimiento para barajar y repartir cartas.

En el programa, la función `llenaMazo` (líneas 44 a 55) inicializa el arreglo `Carta` en orden de As a Rey de cada palo. El arreglo `Carta` se pasa (en la línea 36) a la función `barajar` (líneas 58 a 72), en donde se implementa el algoritmo de alto rendimiento para barajar. La función `barajar` toma como su argumento a un arreglo de estructuras de 52 `Cartas`. La función hace un ciclo a través de las 52 cartas (arreglos con subíndices del 0 al 51), utilizando una instrucción `for` en las líneas 65 a 70. Por cada carta, se escoge al azar un número entre el 0 y el 51. Después, la estructura actual `Carta` y la estructura `Carta` seleccionada al azar se intercambian en el arreglo (líneas 67 a 69). Se realizan un total de 52 intercambios en una sola pasada del arreglo completo, ¡y el arreglo de estructuras `Carta` está barajado! Este algoritmo no puede sufrir un aplazamiento indefinido

como el algoritmo que presentamos en el capítulo 7. Debido a que se intercambiaron las estructuras **`Carta`** en el lugar del arreglo, el algoritmo de alto rendimiento para repartir, implementado en la función repartir (líneas 75 a 85), requiere sólo una pasada en el arreglo para repartir las cartas barajadas.



**Error común de programación 10.8**

*Olvidar incluir el subíndice del arreglo cuando se hace referencia a estructuras individuales del arreglo de estructuras, es un error de sintaxis.*

## 10.8 Uniones

Una *unión* es un tipo de dato derivado, como una estructura, con miembros que comparten el mismo espacio de almacenamiento. Para diferentes situaciones en un programa, algunas variables podrían no ser importantes, pero otras sí; por lo tanto, una unión comparte el espacio, en lugar de desperdiciar espacio en variables que no se están utilizando. Los miembros de una unión pueden ser de cualquier tipo. El número de bytes utilizado para almacenar una unión debe ser suficiente para almacenar al menos el miembro más grande. En la mayoría de los casos, las uniones contienen dos o más tipos de datos. Se puede hacer referencia solamente a un miembro a la vez, y por lo tanto a un tipo de dato a la vez. Es responsabilidad del programador garantizar que se haga referencia a los datos de una unión con el tipo de dato apropiado.



**Error común de programación 10.9**

*Hacer referencia a un dato de una unión por medio de una variable del tipo equivocado, es un error lógico.*



**Tip de portabilidad 10.3**

*Si los datos están almacenados en una unión como un tipo y se hace referencia a ellos como otro tipo, los resultados dependen de la implementación.*

Una unión se declara con la palabra reservada **`union`**, en el mismo formato que una estructura. La definición de **`union`**

```
union numero {
   int x;
   double y;
};
```

indica que **`numero`** es un tipo **`union`** con los miembros **`int x`** y **`double y`**. La definición de una unión normalmente se coloca en un encabezado y se incluye en todos los archivos fuente que utilizan ese tipo de unión.



**Observación de ingeniería de software 10.1**

*Así como en una definición de **`struct`**, una definición de **`union`** simplemente crea un tipo nuevo. Al colocar una definición de una unión o de una estructura fuera de cualquier función, no se genera una variable global.*

Las operaciones que pueden realizarse con una unión son las siguientes: asignación de una unión a otra del mismo tipo, tomar la dirección (**&**) de una variable unión, y acceder a los miembros de la unión a través del operador miembro de la estructura y del operador apuntador de la estructura. Las uniones no deben compararse por medio de los operadores **`==`** y **`!=`**, por las mismas razones que indican que las estructuras no deben compararse.

En una declaración, una unión puede inicializarse con un valor del mismo tipo que el primer miembro de la unión. Por ejemplo, con la unión anterior, la declaración

```
union numero valor = { 10 };
```

es una inicialización válida para una variable unión, ya que ésta se inicializa con un **`int`**, sin embargo, la siguiente declaración truncaría la parte de punto flotante del inicializador, y normalmente produciría una advertencia por parte del compilador:

```
union numero valor = { 1.43 };
```



**Error común de programación 10.10**

*Comparar uniones es un error de sintaxis.*



**Tip de portabilidad 10.4**

*La cantidad de almacenamiento requerida para almacenar una unión depende de la implementación.*



**Tip de portabilidad 10.5**

*Algunas uniones podrían no portarse fácilmente a otros sistemas de cómputo. El que una unión sea portable o no, no siempre depende de los requerimientos de alineación de almacenamiento para los datos miembro de la unión en un sistema dado.*

**Tip de rendimiento 10.2**

*Las uniones conservan el almacenamiento.*

El programa de la figura 10.5 utiliza la variable valor (línea 13) del tipo **`union numero`**, para desplegar el valor almacenado en la unión, tanto como un **`int`** como un **`double`**. La salida del programa depende de la implementación. La salida del programa muestra que la representación interna de un valor **`double`** puede ser muy diferente de la representación del **`int`**.

```
1  /* Figura 10.5: fig10_05.c
2     Un ejemplo de unión */
3  #include <stdio.h>
4
5  /* definición de la unión numero */
6  union numero {
7     int x;
8     double y;
9  }; /* fin de la unión numero */
10
11 int main()
12 {
13    union numero valor; /* define la variable de unión */
14
15    valor.x = 100; /* coloca un entero dentro de la unión */
16    printf( "%s\n%s\n%s%d\n%s%f\n\n",
17       "Coloca un valor en el miembro entero",
18       "e imprime ambos miembros.",
19       "int:   ", valor.x,
20       "double:\n", valor.y );
21
22    valor.y = 100.0; /* coloca un double dentro de la misma unión */
23    printf( "%s\n%s\n%s%d\n%s%f\n",
24       "Coloca un valor en el miembro flotante",
25       "e imprime ambos miembros.",
26       "int:   ", valor.x,
27       "double:\n", valor.y );
28
29    return 0; /* indica terminación exitosa */
30
31 } /* fin de main */
```

```
Coloca un valor en el miembro entero
e imprime ambos miembros.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000.000000
```

**Figura 10.5** Cómo desplegar el valor de una unión en los dos tipos de datos miembro. (Parte 1 de 2.)

```
Coloca un valor en el miembro flotante
e imprime ambos miembros.
int:   0
double:
100.000000
```

**Figura 10.5** Cómo desplegar el valor de una unión en los dos tipos de datos miembro. (Parte 2 de 2.)

## 10.9 Operadores a nivel de bits

Las computadoras representan internamente todos los datos como secuencias de bits. Cada bit puede asumir un valor de **0** o **1**. En la mayoría de los sistemas, una secuencia de 8 bits forma un byte; la unidad estándar de almacenamiento para una variable de tipo **char**. Otros tipos de datos se almacenan en números con más bytes. Los operadores a nivel de bits se utilizan para manipular los bits de operandos enteros (**char**, **short**, **int** y **long**; tanto **signed** como **unsigned**). Los enteros sin signo con frecuencia se utilizan con los operadores a nivel de bits.

**Tip de portabilidad 10.6**

*Las manipulaciones de datos a nivel de bits dependen de la máquina.*

Observe que las explicaciones de esta sección sobre los operadores a nivel de bits muestran la representación binaria de los operandos enteros. Para una explicación detallada de los sistemas binarios de numeración (también llamados de base 2), vea el apéndice E. Además, los programas de la sección 10.9 se evaluaron por medio del Visual C++ de Microsoft. Debido a que la naturaleza dependiente de la máquina de las manipulaciones a nivel de bits, estos programas podrían no funcionar en su sistema.

Los operadores a nivel de bits son: *AND a nivel de bits (***&***)*, *OR incluyente a nivel de bits (***|***)*, *OR excluyente a nivel de bits (***^***)*, *desplazamiento a la izquierda (***<<***)*, *desplazamiento a la derecha (***>>***)* y *complemento (***~***)*. Los operadores a nivel de bits AND, OR incluyente y OR excluyente comparan sus dos operandos bit por bit. El operador AND a nivel de bits establece en 1 cada bit del resultado, si el bit correspondiente a ambos operandos es 1. El operador OR incluyente a nivel de bits establece en 1 cada bit del resultado, si el bit correspondiente a uno o a ambos operandos es 1. El operador OR excluyente a nivel de bits establece en 1 cada bit del resultado, si el bit correspondiente a exactamente un operando es 1. El operador de desplazamiento a la izquierda desplaza hacia la izquierda los bits de su operando izquierdo,  el número de bits especificados en su operando derecho. El operador de desplazamiento a la derecha desplaza hacia la derecha los bits de su operando izquierdo, el número de bits especificados en su operando derecho. El operador de complemento a nivel de bits hace que todos los bits que se encuentran en **0** en su operando se establezcan en **1** en el resultado, y que todos los que se encuentran en **1** en su operando, se establezcan en **0** en el resultado. En los siguientes ejemplos mostramos explicaciones detalladas de cada operador a nivel de bits. La figura 10.6 resume los operadores a nivel de bits.

| Operador | Descripción |
|---|---|
| & AND a nivel de bits | Los bits del resultado se establecen en **1**, si los bits correspondientes a los dos operandos son **1**. |
| \| OR incluyente a nivel de bits | Los bits del resultado se establecen en **1**, si al menos uno de los bits correspondientes a los dos operandos es **1**. |
| ^ OR excluyente a nivel de bits | Los bits del resultado se establecen en **1**, si exactamente uno de los bits correspondientes a los dos operandos es **1**. |
| << desplazamiento a la izquierda | Desplaza hacia la izquierda los bits del primer operando, el número de bits especificados por el segundo operando; desde la derecha llena con bits en **0**. |

**Figura 10.6** Operadores a nivel de bits. (Parte 1 de 2.)

| Operador | Descripción |
|---|---|
| >> desplazamiento a la derecha | Desplaza hacia la derecha los bits del primer operando, el número de bits especificados por el segundo operando; el método de llenado desde la izquierda depende de la máquina. |
| ~ complemento a uno | Todos los bits en **0** se establecen en **1**, y todos los bits en **1** se establecen en **0**. |

**Figura 10.6** Operadores a nivel de bits. (Parte 2 de 2.)

***Cómo desplegar en bits un entero sin signo***

Cuando utilizamos los operadores a nivel de bits es útil imprimir valores en su representación binaria, para ilustrar los efectos precisos de estos operadores. El programa de la figura 10.7 imprime un entero sin signo en su representación binaria, en grupos de ocho bits cada uno.

```
/* Figura 10.7: fig10_07.c
   Impresión en bits de un entero sin signo */
#include <stdio.h>

void despliegaBits( unsigned valor ); /* prototipo */

int main()
{
   unsigned x; /* variable para almacenar la entrada del usuario */

   printf( “Introduzca un entero sin signo: “ );
   scanf( “%u”, &x );

   despliegaBits( x );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* despliega los bits de un valor entero sin signo */
void despliegaBits( unsigned valor )
{
   unsigned c; /* contador */

   /* define despliegaMascara y desplaza 31 bits hacia la izquierda */
   unsigned despliegaMascara = 1 << 31;

   printf( “%10u = “, valor );

   /* ciclo a través de los bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( valor & despliegaMascara ? ‘1’ : ‘0’ );
      valor <<= 1; /* desplaza valor 1  hacia la izquierda */

      if ( c % 8 == 0 ) { /* despliega espacio después de 8 bits */
         putchar( ‘ ‘ );
      } /* fin de if */

```

**Figura 10.7** Cómo desplegar en bits un entero sin signo. (Parte 1 de 2.)

```
39      } /* fin de for */
40
41      putchar( '\n' );
42   } /* fin de la función despliegaBits */
```

```
Introduzca un entero sin signo: 65000
     65000 = 00000000 00000000 11111101 11101000
```

**Figura 10.7** Cómo desplegar en bits un entero sin signo. (Parte 2 de 2.)

La función **despliegaBits** (líneas 21 a 42) utiliza el operador AND a nivel de bits para combinar la variable **valor** con la variable **despliegaMascara** (línea 32). Con frecuencia, el operador AND a nivel de bits se utiliza con un operando llamado *máscara*; un valor entero con bits específicos establecidos en **1**. Las máscaras se utilizan para ocultar algunos bits en un valor, mientras se seleccionan otros bits. En la función **despliegaBits**, a la variable máscara **despliegaMascara** se le asigna el valor

**1 << 31 (10000000 00000000 00000000 00000000)**

El operador de desplazamiento a la izquierda cambia el valor **1** de un orden bajo de bit (el que se encuentra más hacia la derecha) a un orden más alto de bit (más hacia la izquierda) en **despliegaMascara**, y llena con bits en **0** los espacios desde la derecha. La línea 32

```
putchar( valor & despliegaMascara ? '1' : '0' );
```

determina si debe imprimirse un **1** o un **0** para el bit actual más a la izquierda de la variable valor. Cuando **valor** y **despliegaMascara** se combinan por medio de **&**, todos los bits en la variable **valor**, excepto el bit de mayor orden, se "enmascaran" (se ocultan), debido a que cualquier bit en **0** al que se le aplique el operador AND, arrojará **0**. Si el bit más a la izquierda es **1**, **valor&despliegaMascara** da como resultado un valor diferente de cero (verdadero), y se imprime un **1**; de lo contrario, se imprime un **0**. Después, la variable **valor** se desplaza un bit hacia la izquierda, por medio de la expresión **valor <<= 1** (esto es equivalente a **valor = valor << 1**). Estos pasos se repiten para cada bit de la variable **unsigned valor**. La figura 10.8 resume los resultados de combinar dos bits con el operador AND a nivel de bits.



**Error común de programación 10.11**

*Utilizar el operador lógico AND (&&) en lugar del operador a nivel de bits AND (&), y viceversa, es un error.*

***Cómo utilizar los operadores a nivel de bits AND, OR incluyente, OR excluyente y complemento***

La figura 10.9 muestra el uso de los operadores a nivel de bits AND, OR incluyente, OR excluyente y el de complemento. El programa utiliza la función **despliegaBits** (líneas 53 a 74) para imprimir los valores enteros **unsigned**. La salida aparece en la figura 10.10.

En la figura 10.9, en la línea 16, a la variable entera **numero1** se le asigna el valor **65535** (**00000000 00000000 11111111 11111111**) y, en la línea 17, a la variable **mascara** se le asigna el valor de **1** (**00000000 00000000 00000000 00000001**). Cuando **numero1** y **mascara** se combinan por medio del operador a nivel de bits AND (&) en la expresión **numero1&mascara** (línea 22), el resultado es

| Bit 1 | Bit 2 | Bit 1 & Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

**Figura 10.8** Resultados de combinar dos bits con el operador a nivel de bits AND (**&**).

```
/* Figura 10.9: fig10_09.c
   Uso de los operadores de bits AND, OR incluyente,
   OR excluyente a nivel de bits y complemento */
#include <stdio.h>

void despliegaBits( unsigned valor ); /* prototipo */

int main()
{
   unsigned numero1; /* define numero1 */
   unsigned numero2; /* define numero2 */
   unsigned mascara; /* define mascara */
   unsigned estableceBits; /* define estableceBits */

   /* demuestra el operador de bits AND (&) */
   numero1 = 65535;
   mascara = 1;
   printf( "El resultado de combinar los siguientes valores\n" );
   despliegaBits( numero1 );
   despliegaBits( mascara );
   printf( "con el uso del operador de bits AND (&) es\n" );
   despliegaBits( numero1 & mascara );

   /* demuestra el operador de bits OR incluyente (|) */
   numero1 = 15;
   estableceBits = 241;
   printf( "\nEl resultado de combinar los siguientes valores\n" );
   despliegaBits( numero1 );
   despliegaBits( estableceBits );
   printf( "con el uso del operador de bits OR incluyente (|) es\n" );
   despliegaBits( numero1 | estableceBits );

   /* demuestra el operador de bits OR excluyente (^) */
   numero1 = 139;
   numero2 = 199;
   printf( "\nEl resultado de combinar los siguientes valores\n" );
   despliegaBits( numero1 );
   despliegaBits( numero2 );
   printf( "con el uso del operador de bits OR excluyente (^) es\n" );
   despliegaBits( numero1 ^ numero2 );

   /* demuestra el operador de bits complemento (~) */
   numero1 = 21845;
   printf( "\nEl complemento a uno de\n" );
   despliegaBits( numero1 );
   printf( "es\n" );
   despliegaBits( ~numero1 );

   return 0; /* indica terminación exitosa */
} /* fin de main */

/* despliega los bits de un valor entero sin signo */
void despliegaBits( unsigned valor )
{
   unsigned c; /* contador */

```

**Figura 10.9** Operadores a nivel de bits AND, OR incluyente, OR excluyente y complemento. (Parte 1 de 2.)

```
57      /* declara despliegaMarcara y desplaza 31 bits a la izquierda */
58      unsigned despliegaMascara = 1 << 31;
59
60      printf( "%10u = ", valor );
61
62      /* ciclo a través de los bits */
63      for ( c = 1; c <= 32; c++ ) {
64       putchar( valor & despliegaMascara ? '1' : '0' );
65       valor <<= 1; /* desplaza el valor 1 bit a la izquierda */
66
67         if ( c % 8 == 0 ) { /* muestra un espacio después de 8 bits */
68          putchar( ' ' );
69         } /* fin de if */
70
71      } /* fin de for */
72
73      putchar( '\n' );
74   } /* fin de la función despliegaBits */
```

**Figura 10.9** Operadores a nivel de bits AND, OR incluyente, OR excluyente y complemento. (Parte 2 de 2.)

```
El resultado de combinar los siguientes valores
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
con el uso del operador de bits AND & es
         1 = 00000000 00000000 00000000 00000001

El resultado de combinar los siguientes valores
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
con el uso del operador de bits OR incluyente | es
       255 = 00000000 00000000 00000000 11111111

El resultado de combinar los siguientes valores
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
con el uso del operador de bits OR excluyente (^) es
        76 = 00000000 00000000 00000000 01001100

El complemento a uno de
     21845 = 00000000 00000000 01010101 01010101
es
4294945450 = 11111111 11111111 10101010 10101010
```

**Figura 10.10** Salida del programa correspondiente a la figura 10.9.

**00000000 00000000 00000000 00000001**. Todos los bits de la variable **numero1**, excepto el de orden más bajo, se enmascaran (se ocultan) al aplicarles el operador AND con la variable **mascara**.

El operador a nivel de bits OR incluyente se utiliza para establecer en un operando bits específicos en 1. En la figura 10.9, en la línea 25, a la variable **numero1** se le asigna **15** (**00000000 00000000 00000000 00001111**) y, en la línea 26, a la variable **estableceBits** se le asigna **241** (**00000000 00000000 00000000 11110001**). Cuando **numero1** y **estableceBits** se combinan por medio del operador a nivel de bits OR, en la expresión **numero1 | estableceBits** (línea 31), el resultado es **255** (**00000000 00000000 00000000 11111111**). La figura 10.11 resume los resultados de combinar dos bits mediante el operador a nivel de bits OR incluyente.

| Bit 1 | Bit 2 | Bit 1 \| Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

**Figura 10.11** Resultados de combinar dos bits con el operador a nivel de bits OR incluyente (**|**).



**Error común de programación 10.12**

*Utilizar el operador lógico OR (**||**) en lugar del operador a nivel de bits OR (**|**), y viceversa, es un error.*

El operador a nivel de bits OR excluyente (**^**) establece cada bit del resultado en 1, si *exactamente* uno de los operadores correspondientes a los bits de sus dos operandos es 1. En la figura 10.9, en las líneas 34 y 35, a las variables **numero1** y **numero2** se les asignan los valores **139** (**00000000 00000000 00000000 10001011**) y **199** (**00000000 00000000 00000000 11000111**). Cuando estas variables se combinan por medio del operador OR excluyente, en la expresión **numero1^numero2** (línea 40), el resultado es **00000000 00000000 00000000 01001100**. La figura 10.12 resume los resultados de combinar dos bits con el operador a nivel de bits OR excluyente.

El operador de complemento *a nivel de bits* (**~**) hace que todos los bits que se encuentran en **0** en su operando se establezcan en **1** en el resultado, y que todos los que se encuentran en **1** en su operando, se establezcan en **0** en el resultado; otro modo de decir esto es "tomar el *complemento a uno* del valor". En la figura 10.9, en la línea 43, a la variable **numero1** se le asigna el valor **21845** (**00000000 00000000 01010101 01010101**). Cuando se evalúa la expresión **~numero1** (línea 47), el resultado es **00000000 00000000 10101010 10101010**.

***Cómo utilizar los operadores a nivel de bits de desplazamiento a la izquierda y de desplazamiento a la derecha***

El programa de la figura 10.13 muestra los operadores de desplazamiento a la izquierda (**<<**) y de desplazamiento a la derecha (**>>**). La función **despliegaBits** se utiliza para imprimir los valores enteros **unsigned**.

| Bit 1 | Bit 2 | Bit 1 ^ Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

**Figura 10.12** Resultados de combinar dos bits con el operador a nivel de bits OR excluyente (**^**).

```
/* Figura 10.13: fig10_13.c
   Uso de los operadores de desplazamiento de bits */
#include <stdio.h>

void despliegaBits( unsigned valor ); /* prototipo */

int main()
{
   unsigned numero1 = 960; /* inicializa numero1 */
```

**Figura 10.13** Operadores de desplazamiento a nivel de bits. (Parte 1 de 2.)

```
   /* demuestra el operador de desplazamiento a la izquierda a nivel de bits */
   printf( "\nEl resultado del desplazamiento a la izquierda de\n" );
   despliegaBits( numero1 );
   printf( "8 posiciones de bit con el uso del " );
   printf( "operador de desplazamiento a la izquierda << es\n" );
   despliegaBits( numero1 << 8 );

   /* demuestra el operador de desplazamiento a la derecha a nivel de bits */
   printf( "\nEl resultado del desplazamiento a la derecha de\n" );
   despliegaBits( numero1 );
   printf( "8 posiciones de bit con el uso del " );
   printf( "operador de desplazamiento a la derecha >> es\n \n" );
   despliegaBits( numero1 >> 8 );

   return 0; /* indica terminación exitosa */
} /* fin de main */

/* despliega los bits de un valor entero sin signo */
void despliegaBits( unsigned valor )
{
   unsigned c; /* contador */

   /* declara despliegaMascara y desplaza a la izquierda 31 bits */
   unsigned despliegaMascara = 1 << 31;

   printf( "%7u = ", valor );

   /* ciclo a través de los bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( valor & despliegaMascara ? '1' : '0' );
      valor <<= 1; /* despliega el valor 1 posición a la izquierda */

      if ( c % 8 == 0 ) { /* muestra un espacio después de 8 bits */
         putchar( ' ' );
      } /* fin de if */

   } /* fin de for */

   putchar( '\n' );
} /* fin de la función despliegaBits */
```

```
El resultado del desplazamiento a la izquierda de
    960 = 00000000 00000000 00000011 11000000
8 posiciones de bit con el uso del operador de desplazamiento a la izquierda <<
es
 245760 = 00000000 00000011 11000000 00000000

El resultado del desplazamiento a la derecha de
    960 = 00000000 00000000 00000011 11000000
8 posiciones de bit con el uso del operador de desplazamiento a la derecha >> es
      3 = 00000000 00000000 00000000 00000011
```

**Figura 10.13** Operadores de desplazamiento a nivel de bits. (Parte 2 de 2.)

El operador de desplazamiento a la izquierda (**<<**) desplaza hacia la izquierda los bits de su operando izquierdo, el número de bits especificados en su operando derecho. Los bits desocupados se reemplazan con ceros; los unos desplazados hacia la izquierda se pierden. En la figura 10.13, en la línea 9, a la variable **numero1** se le asigna el valor **960** (**00000000 00000000 00000011 11000000**). El resultado de desplazar 8 bits hacia la izquierda a la variable **numero1**, por medio de la expresión **numero1<<8** (línea 16) es **49152** (**00000000 00000000 11000000 00000000**).

El operador de desplazamiento a la derecha (**>>**) desplaza hacia la derecha los bits de su operando izquierdo, el número de bits especificado en su operando derecho. Aplicar un desplazamiento hacia la derecha sobre un entero **unsigned** ocasiona que los bits desocupados sean reemplazados con ceros; los unos desplazados hacia la derecha se pierden. En la figura 10.13, el resultado de desplazar hacia la derecha **numero1** por medio de la expresión **numero1>>8** (línea 23) es **3** (**00000000 00000000 00000000 00000011**).



**Error común de programación 10.13**

*El resultado de desplazar un valor es indefinido, si el operando derecho es negativo o si es mayor que el número de bits en el que el operando izquierdo está almacenado.*



**Tip de portabilidad 10.7**

*El desplazamiento a la derecha es dependiente de la máquina. Aplicar un desplazamiento a la derecha a un entero con signo en algunas máquinas ocasiona que los bits desocupados se llenen con ceros, y en otras que se llenen con unos.*

### *Operadores de asignación a nivel de bits*

Todo operador binario a nivel de bits tiene un operador de asignación correspondiente. Estos *operadores de asignación a nivel de bits* aparecen en la figura 10.14 y se utilizan de manera similar a los operadores aritméticos de asignación que presentamos en el capítulo 3.

La figura 10.15 muestra la precedencia y asociatividad de los diversos operadores que hemos presentado hasta este punto. Éstos aparecen de arriba hacia abajo, en orden decreciente de precedencia.

| Operadores de asignación a nivel de bits | |
|---|---|
| **&=** | Operador de asignación AND a nivel de bits. |
| **\|=** | Operador de asignación OR incluyente a nivel de bits. |
| **^=** | Operador de asignación OR excluyente a nivel de bits. |
| **<<=** | Operador de asignación de desplazamiento a la izquierda. |
| **>>=** | Operador de asignación de desplazamiento a la derecha. |

**Figura 10.14** Operadores de asignación a nivel de bits.

| Operador | Asociatividad | Tipo |
|---|---|---|
| **( ) [ ] . ->** | izquierda a derecha | el más alto |
| **+ - ++ -- ! & * ~ sizeof** (*tipo*) | derecha a izquierda | unario |
| *** / %** | izquierda a derecha | de multiplicación |
| **+ -** | izquierda a derecha | de adición |
| **<< >>** | izquierda a derecha | de desplazamiento |
| **< <= > >=** | izquierda a derecha | de relación |
| **== !=** | izquierda a derecha | de igualdad |
| **&** | izquierda a derecha | AND a nivel de bits |
| **^** | izquierda a derecha | OR a nivel de bits |

**Figura 10.15** Precedencia y asociatividad de operadores. (Parte 1 de 2.)

| Operador | Asociatividad | Tipo |
|---|---|---|
| `\|` | izquierda a derecha | OR a nivel de bits |
| `&&` | izquierda a derecha | AND lógico |
| `\|\|` | izquierda a derecha | OR lógico |
| `?:` | derecha a izquierda | condicional |
| `= += -= *= /= &= \|= ^= <<= >>= %=` | derecha a izquierda | de asignación |
| `,` | izquierda a derecha | coma |

**Figura 10.15** Precedencia y asociatividad de operadores. (Parte 2 de 2.)

## 10.10 Campos de bits

C permite a los programadores especificar el número de bits en el que un miembro **`unsigned`** o **`int`** de una estructura o unión se almacena. A esto se le conoce como *campo de bits*. Los campos de bits permiten una mejor utilización de la memoria, ya que almacenan los datos en el número mínimo de bits necesario. Los miembros de un campo de bits deben declararse como **`int`** o **`unsigned`**.



**Tip de rendimiento 10.3**

*Los campos de bits ayudan a conservar el almacenamiento.*

Considere la siguiente definición de una estructura:

```
struct cartaBit {
   unsigned cara : 4;
   unsigned palo : 2;
  unsigned color : 1;
};
```

La definición contiene tres campos de bits **`unsigned`** (**`cara`**, **`palo`** y **`color`**) que se utilizan para representar una carta de un mazo de 52 cartas. Un campo de bits se declara colocando un *nombre de miembro* **`unsigned`** o **`int`** seguido por dos puntos (**`:`**) y una constante entera que representa el *ancho* del campo (es decir, el número de bits en el que se almacena el miembro). La constante que representa el ancho debe ser un entero entre **`0`** y el número total de bits utilizado para almacenar un **`int`** en su sistema. Nuestros ejemplos se evaluaron en una computadora con enteros de 4 bytes (32 bits).

La definición anterior indica que el miembro **`cara`** se almacena en 4 bits, el miembro **`palo`** en 2 bits y el miembro **`color`** en 1 bit. El número de bits se basa en el rango deseado de valores para cada miembro de la estructura. El miembro **`cara`** almacena valores del **`0`** (As) al **`12`** (Rey); 4 bits pueden almacenar valores en el rango del **`0`** al **`15`**. El miembro **`palo`** almacena valores del **`0`** al **`3`** (**`0`** = Diamantes, **`1`** = Corazones, **`2`** = Tréboles, **`3`** = Espadas); 2 bits pueden almacenar valores en el rango del **`0`** al **`3`**. Por último, el miembro **`color`** almacena **`0`** (Rojo) o **`1`** (Negro); 1 bit puede almacenar **`0`** o **`1`**.

La figura 10.16 (cuya salida aparece en la figura 10.17), en la línea 20, crea un arreglo **`mazo`** que contiene 52 estructuras **`struct cartaBit`**. La función **`llenaMazo`** (líneas 30 a 41) inserta las 52 cartas en el arreglo **`mazo`**, y la función **`repartir`** (líneas 45 a 58) imprime las 52 cartas. Observe que se accede a los miembros del campo de bits de la misma manera que con cualquier otra estructura. El miembro **`color`** se incluye como un medio para indicar el color de una carta en un sistema que permite el despliegue de color.

```
1  /* Figura 10.16: fig10_16.c
2     Representación de barajas mediante campos de bits en una estructura */
3
4  #include <stdio.h>
```

**Figura 10.16** Campos de bits para almacenar un mazo de cartas. (Parte 1 de 2.)

```

/* definición de la estructura cartaBit con campos de bits */
struct cartaBit {
   unsigned cara : 4;  /* 4 bits; 0-15 */
   unsigned palo : 2;  /* 2 bits; 0-3 */
   unsigned color : 1; /* 1 bit; 0-1 */
}; /* fin de la estructura cartaBit */

typedef struct cartaBit Carta; /* nuevo nombre de tipo para la estructura
                                  cartaBit */

void llenaMazo( Carta * const wMazo );       /* prototipo */
void repartir( const Carta * const wMazo ); /* prototipo */

int main()
{
   Carta mazo[ 52 ]; /* crea el arreglo de Cartas */

   llenaMazo( mazo );
   repartir( mazo );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* inicializa Carta */
void llenaMazo( Carta * const wMazo )
{
   int i; /* contador */

   /* ciclo a través de wMazo */
   for ( i = 0; i <= 51; i++ ) {
      wMazo[ i ].cara = i % 13;
      wMazo[ i ].palo = i / 13;
      wMazo[ i ].color = i / 26;
   } /* fin de for */

} /* fin de la función llenaMazo */

/* muestra las cartas en formato de dos columnas; el subíndice de las
   cartas 0 a 25 es k1 (columna 1); el subíndice de las cartas 26 a 51
   es k2 (columna 2) */
void repartir( const Carta * const wMazo )
{
   int k1; /* subíndice 0-25 */
   int k2; /* subíndice 26-51 */

   /* ciclo a través de wMazo */
   for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      printf( “Carta:%3d  Palo:%2d  Color:%2d   “,
         wMazo[ k1 ].cara, wMazo[ k1 ].palo, wMazo[ k1 ].color );
      printf( “Carta:%3d  Palo:%2d  Color:%2d\n”,
         wMazo[ k2 ].cara, wMazo[ k2 ].palo, wMazo[ k2 ].color );
   } /* fin de for */

} /* fin de la función repartir */
```

**Figura 10.16** Campos de bits para almacenar un mazo de cartas. (Parte 2 de 2.)

```
Carta:  0  Palo: 0  Color: 0   Carta:  0  Palo: 2  Color: 1
Carta:  1  Palo: 0  Color: 0   Carta:  1  Palo: 2  Color: 1
Carta:  2  Palo: 0  Color: 0   Carta:  2  Palo: 2  Color: 1
Carta:  3  Palo: 0  Color: 0   Carta:  3  Palo: 2  Color: 1
Carta:  4  Palo: 0  Color: 0   Carta:  4  Palo: 2  Color: 1
Carta:  5  Palo: 0  Color: 0   Carta:  5  Palo: 2  Color: 1
Carta:  6  Palo: 0  Color: 0   Carta:  6  Palo: 2  Color: 1
Carta:  7  Palo: 0  Color: 0   Carta:  7  Palo: 2  Color: 1
Carta:  8  Palo: 0  Color: 0   Carta:  8  Palo: 2  Color: 1
Carta:  9  Palo: 0  Color: 0   Carta:  9  Palo: 2  Color: 1
Carta: 10  Palo: 0  Color: 0   Carta: 10  Palo: 2  Color: 1
Carta: 11  Palo: 0  Color: 0   Carta: 11  Palo: 2  Color: 1
Carta: 12  Palo: 0  Color: 0   Carta: 12  Palo: 2  Color: 1
Carta:  0  Palo: 1  Color: 0   Carta:  0  Palo: 3  Color: 1
Carta:  1  Palo: 1  Color: 0   Carta:  1  Palo: 3  Color: 1
Carta:  2  Palo: 1  Color: 0   Carta:  2  Palo: 3  Color: 1
Carta:  3  Palo: 1  Color: 0   Carta:  3  Palo: 3  Color: 1
Carta:  4  Palo: 1  Color: 0   Carta:  4  Palo: 3  Color: 1
Carta:  5  Palo: 1  Color: 0   Carta:  5  Palo: 3  Color: 1
Carta:  6  Palo: 1  Color: 0   Carta:  6  Palo: 3  Color: 1
Carta:  7  Palo: 1  Color: 0   Carta:  7  Palo: 3  Color: 1
Carta:  8  Palo: 1  Color: 0   Carta:  8  Palo: 3  Color: 1
Carta:  9  Palo: 1  Color: 0   Carta:  9  Palo: 3  Color: 1
Carta: 10  Palo: 1  Color: 0   Carta: 10  Palo: 3  Color: 1
Carta: 11  Palo: 1  Color: 0   Carta: 11  Palo: 3  Color: 1
Carta: 12  Palo: 1  Color: 0   Carta: 12  Palo: 3  Color: 1
```

**Figura 10.17** Salida del programa de la figura 10.16.

Es posible especificar un *campo de bits sin nombre* para utilizarlo como *relleno* en la estructura. Por ejemplo, la definición de la estructura

```
struct ejemplo {
   unsigned a : 13;
   unsigned   : 19;
   unsigned b : 4;
};
```

utiliza un campo sin nombre de 19 bits como relleno; nada puede almacenarse en esos 19 bits. El miembro **b** (en nuestra computadora de palabras de 4 bytes) se almacena en otra unidad de almacenamiento.

Un *campo de bits sin nombre con un ancho de 0* se utiliza para alinear el siguiente campo de bits en un nuevo límite de la unidad de almacenamiento. Por ejemplo, la definición de la estructura

```
struct ejemplo {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

utiliza un campo de bits sin nombre de **0** bits, para saltar los bits restantes (todos los que hayan) de la unidad de almacenamiento en la que se almacena **a**, y para alinear **b** son el siguiente límite de la unidad de almacenamiento.

**Tip de portabilidad 10.8**

*Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten campos de bits que crucen los límites de palabras, y otras no lo hacen.*



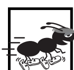
**Error común de programación 10.14**

*Intentar acceder a bits individuales de un campo de bits, como si fueran elementos de un arreglo, es un error de sintaxis. Los campos de bits no son "arreglos de bits".*



**Error común de programación 10.15**

*Intentar tomar la dirección de un campo de bits (el operador **&** no debe utilizarse con campos de bits, ya que éstos no tienen direcciones).*



**Tip de rendimiento 10.4**

*Aunque los campos de bits ahorran espacio, utilizarlos puede ocasionar que el compilador genere código en lenguaje máquina de ejecución lenta. Esto ocurre debido a que éste toma operaciones adicionales en lenguaje máquina para acceder sólo a porciones de una unidad de almacenamiento direccionable. Éste es uno de los muchos ejemplos del equilibrio espacio-tiempo que se suscitan en la ciencia de la computación.*

## 10.11 Constantes de enumeración

C proporciona un último tipo definido por el usuario llamado *enumeración*. Una enumeración, introducida por medio de la palabra reservada **enum**, es un conjunto entero de *constantes de enumeración* representadas mediante identificadores. Los valores en una **enum** comienzan con **0**, a menos que se especifique lo contrario, y se incrementan en **1**. Por ejemplo, la enumeración

```
enum meses {
   ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC };
```

crea un nuevo tipo, **enum meses**, en el que los identificadores se establecen en los enteros de **0** a **11**, respectivamente. Para numerar los meses 1 a 12, utilice la siguiente enumeración:

```
enum meses {
   ENE = 1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC };
```

Debido a que el primer valor de la enumeración anterior se establece explícitamente en **1**, los valores restantes se incrementan a partir de **1**, lo que da como resultado los valores del **1** al **12**. Los identificadores de una enumeración deben ser únicos. El valor de cada constante de enumeración puede establecerse explícitamente en la definición, asignándole un valor al identificador. Diversos miembros de una enumeración pueden tener el mismo valor constante. En el programa de la figura 10.18, la variable de enumeración **mes** se utiliza en una instrucción **for** para imprimir los meses, a partir del arreglo **nombreMes**. Observe que hemos hecho que **nombreMes[ 0 ]** sea la cadena vacía **“ ”**. Algunos programadores podrían preferir establecer **nombreMes[ 0 ]** en un valor como *****ERROR*****, para indicar que ocurrió un error lógico.

```
/* Figura 10.18: fig10_18.c
   Uso de un tipo de enumeración */
#include <stdio.h>

/* las constantes de enumeración representan los meses del año */
enum meses {
   ENE = 1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC };

int main()
{
   enum meses mes; /* puede contener cualquiera de los 12 meses */

   /* inicializa el arreglo de apuntadores  */
   const char *nombreMes[] = { “”, “Enero”, “Febrero”, “Marzo”,
      “Abril”, “Mayo”, “Junio”, “Julio”, “Agosto”, “Septiembre”, “Octubre”,
      “Noviembre”, “Diciembre” };

   /* ciclo a través de los meses  */
   for ( mes = ENE; mes <= DIC; mes++ ) {
      printf( “%2d%11s\n”, mes, nombreMes[ mes ] );
```

**Figura 10.18** Uso de una enumeración. (Parte 1 de 2.)

```
    } /* fin de for */

    return 0; /* indica terminación exitosa */
} /* fin de main */
```

```
 1       Enero
 2     Febrero
 3       Marzo
 4       Abril
 5        Mayo
 6       Junio
 7       Julio
 8      Agosto
 9  Septiembre
10     Octubre
11   Noviembre
12   Diciembre
```

**Figura 10.18** Uso de una enumeración. (Parte 2 de 2.)



### Error común de programación 10.16

*Asignar un valor a una constante de enumeración después de que se definió, es un error de sintaxis.*



### Buena práctica de programación 10.5

*Utilice sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que dichas constantes resalten en un programa, y recuerda al programador que las constantes de enumeración no son variables.*

## RESUMEN

- Las estructuras, algunas veces llamadas agregados, son colecciones de variables relacionadas bajo un mismo nombre.
- Las estructuras pueden contener variables de diferentes tipos de datos.
- La palabra reservada **`struct`** comienza toda definición de estructura. Dentro de las llaves de una definición de estructura se encuentran las declaraciones de los miembros de la estructura.
- Los miembros de la misma estructura deben tener nombres únicos.
- Una definición de estructura crea un nuevo tipo de dato que puede utilizarse para definir variables.
- Existen dos métodos para definir variables de estructuras. El primer método es definir las variables como se hace con las variables de otros tipos de datos, por medio del tipo **`struct etiquetaNombre`**. El segundo método consiste en incluir las variables entre la llave que cierra la definición de la estructura y el punto y coma que finaliza la definición de la estructura.
- La etiqueta con el nombre de la estructura es opcional. Si la estructura se define sin una etiqueta, las variables del tipo de datos derivado debe definirse en la definición de la estructura, y no pueden definirse otras variables de un nuevo tipo de estructura.
- Una estructura puede inicializarse con una lista de inicializadores, colocando después del nombre de la variable un signo de igual y una lista de inicializadores separados por comas, encerrada entre llaves. Si hay menos inicializadores en la lista que miembros en la estructura, los miembros restantes se inicializan automáticamente en cero (o **`NULL`**, si el miembro es un apuntador).
- Estructuras completas pueden asignarse a variables de estructuras del mismo tipo.
- Una variable de estructura puede inicializarse con una variable de estructura del mismo tipo.
- El operador miembro de la estructura se utiliza cuando se accede a un miembro de la estructura, a través del nombre de la variable de estructura.
- El operador apuntador de la estructura (**`->`**), creado con un signo menos (**`-`**) y un signo de mayor que (**`>`**), se utiliza cuando se accede a un miembro de la estructura a través de un apuntador a la estructura.
- Las estructuras y los miembros individuales de las estructuras se pasan por valor a las funciones.

- Para pasar por referencia una estructura, pase la dirección de la variable estructura.
- Un arreglo de estructuras se pasa automáticamente por referencia.
- Para pasar un arreglo por valor, genere una estructura con el arreglo como miembro.
- Al crear un nuevo nombre con **typedef**, no se crea un nuevo tipo; éste crea un nombre que es un sinónimo del tipo definido previamente.
- Una unión es un tipo de dato derivado con miembros que comparten el mismo espacio de almacenamiento. Los miembros pueden ser de cualquier tipo.
- El espacio reservado para una unión es lo suficientemente grande para almacenar su miembro más grande. En la mayoría de los casos, las uniones contienen variables de dos o más tipos. Sólo se puede hacer referencia a un miembro, y por lo tanto a un tipo de dato, a la vez.
- Una unión se declara mediante la palabra reservada **union**, en el mismo formato que una estructura.
- Una unión puede inicializarse con un valor del tipo de su primer miembro.
- El operador a nivel de bits AND (**&**) toma dos operandos integrales. Un bit del resultado se establece en 1, si los bits correspondientes a cada uno de los operandos son 1.
- Las máscaras se utilizan para ocultar algunos bits, mientras se preservan otros.
- El operador a nivel de bits OR incluyente (**|**) toma dos operandos. Un bit en el resultado se establece en 1, si el bit correspondiente a cualquiera de sus operandos se establece en 1.
- Cada uno de los operadores binarios a nivel de bits tiene un operador de asignación correspondiente.
- El operador a nivel de bits OR excluyente (**^**) toma dos operandos. Un bit del resultado se establece en 1, si exactamente uno de los bits correspondientes a los dos operandos se establece en 1.
- El operador de desplazamiento a la izquierda (**<<**) desplaza hacia la izquierda a los bits de su operando izquierdo, el número de bits especificado por su operando derecho. Los bits desocupados a la derecha se reemplazan con ceros.
- El operador de desplazamiento a la derecha (**>>**) desplaza hacia la derecha a los bits de su operando izquierdo, el número de bits especificado por su operando derecho. Realizar un desplazamiento a la derecha sobre un entero sin signo ocasiona que los bits desocupados a la izquierda se reemplazan con ceros. Los bits desocupados en enteros con signo pueden reemplazarse con ceros o unos; esto depende de la máquina.
- El operador a nivel de bits de complemento (**~**) toma un operando e invierte sus bits; esto produce el complemento en unos del operando.
- Los campos de bits reducen el espacio utilizado, almacenando los datos en el número mínimo de bits necesarios.
- Los miembros de un campo de bits deben declararse como **int** o **unsigned**.
- Un campo de bits se declara colocando el nombre de un miembro **int** o **unsigned** seguido por dos puntos y el ancho del campo de bits.
- El ancho de un campo de bits debe ser una constante entera entre 0 y el número total de bits utilizado para almacenar una variable **int** en su sistema.
- Si un campo de bits se especifica sin un nombre, el campo se utiliza como relleno en la estructura.
- Un campo de bits sin nombre con un ancho **0**, alinea el siguiente campo de bits en un nuevo límite de palabras.
- Una enumeración, designada por la palabra reservada **enum**, es un conjunto de enteros que se representan por medio de identificadores. Los valores de una **enum** inician con **0**, a menos que se especifique lo contrario, y se incrementan en **1**.

## TERMINOLOGÍA

acceso a miembros de estructuras
agregados
ancho de un campo de bits
apuntador a una estructura
arreglo de estructuras
asignación de estructuras
campo de bits
campo de bits de ancho cero
campo de bits sin nombre
complemento
complemento en uno
constante de enumeración
declaración de estructuras
definición de estructuras
desplazamiento
desplazamiento a la derecha
desplazamiento a la izquierda
enmascaramiento de bits
enumeración
equilibrio espacio-tiempo
estructura
estructura autorreferenciada
estructuras anidadas
etiqueta con nombre
etiqueta de la estructura
inicialización de una estructura
máscara
miembro
nombre de la estructura
nombre de un miembro
operador (flecha) apuntador de la estructura (**->**)
operador (punto) miembro de la estructura (**.**)
operador a nivel de bits
operador a nivel de bits AND **&**

operador a nivel de bits OR excluyente **^**
operador a nivel de bits OR incluyente **|**
operador de asignación a nivel de bits AND **&=**
operador de asignación de desplazamiento a la derecha **>>=**
operador de asignación de desplazamiento a la izquierda **<<=**
operador de asignación OR excluyente a nivel de bits **^=**
operador de asignación OR incluyente a nivel de bits **|=**
operador de complemento en uno **~**
operador de desplazamiento a la derecha **>>**
operador de desplazamiento a la izquierda **<<**
relleno
**struct**
tipo de dato derivado
tipo de estructura
tipos de datos definidos por el programador
**typedef**
**union**

## *ERRORES COMUNES DE PROGRAMACIÓN*

**10.1** Olvidar el punto y coma al finalizar la definición de una estructura, es un error de sintaxis.

**10.2** Asignar una estructura de un tipo a una estructura de diferente tipo, es un error de compilación.

**10.3** Comparar estructuras es un error de sintaxis.

**10.4** Insertar un espacio entre los componentes **-** y **>** del operador apuntador de la estructura (o insertar espacios entre los componentes de cualquier otro operador con combinación de teclas, excepto **?:**), es un error de sintaxis.

**10.5** Intentar hacer referencia a un miembro de una estructura utilizando únicamente el nombre del miembro, es un error de sintaxis.

**10.6** No utilizar paréntesis cuando se hace referencia al miembro de una estructura que utiliza un apuntador y el operador miembro de la estructura (por ejemplo, ***ptrCarta.palo**), es un error de sintaxis.

**10.7** Suponer que las estructuras, al igual que los arreglos, se pasan automáticamente por referencia, e intentar modificar los valores de la estructura que llama en la función llamada, es un error lógico.

**10.8** Olvidar incluir el subíndice del arreglo cuando se hace referencia a estructuras individuales del arreglo de estructuras, es un error de sintaxis.

**10.9** Hacer referencia a un dato de una unión por medio de una variable del tipo equivocado, es un error lógico.

**10.10** Comparar uniones es un error de sintaxis.

**10.11** Utilizar el operador lógico AND (**&&**) en lugar del operador a nivel de bits AND (**&**), y viceversa, es un error.

**10.12** Utilizar el operador lógico OR (**||**) en lugar del operador a nivel de bits OR (**|**), y viceversa, es un error.

**10.13** El resultado de desplazar un valor es indefinido, si el operando derecho es negativo o si es mayor que el número de bits en el que el operando izquierdo está almacenado.

**10.14** Intentar acceder a bits individuales de un campo de bits, como si fueran elementos de un arreglo, es un error de sintaxis. Los campos de bits no son "arreglos de bits".

**10.15** Intentar tomar la dirección de un campo de bits (el operador **&** no debe utilizarse con campos de bits, ya que éstos no tienen direcciones).

**10.16** Asignar un valor a una constante de enumeración después de que se definió, es un error de sintaxis.

## *TIP PARA PREVENIR ERRORES*

**10.1** Evite utilizar los mismos nombres para los miembros de estructuras de diferentes tipos. Esto está permitido, sin embargo, puede ocasionar confusión.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**10.1** Cuando genere un tipo de estructura, siempre proporcione una etiqueta con su nombre. Dicha etiqueta es conveniente para que posteriormente se declaren nuevas variables correspondientes a la estructura.

**10.2** Elegir una etiqueta con un nombre significativo ayuda a que un programa esté autodocumentado.

**10.3** No coloque espacios alrededor de los operadores (**->**) y (**.**). Omitir los espacios ayuda a enfatizar que las expresiones en las que están contenidos los operadores son esencialmente nombres de variables.

**10.4** Escriba en mayúscula la primera letra de los nombres de **typedef** para enfatizar que esos nombres son sinónimos de los otros nombres de tipos.

**10.5** Utilice sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que dichas constantes resalten en un programa, y recuerda al programador que las constantes de enumeración no son variables.

## TIPS DE RENDIMIENTO

**10.1** Pasar estructuras por referencia resulta más eficiente que pasarlas por valor (lo cual requiere que se copie la estructura completa).

**10.2** Las uniones conservan el almacenamiento.

**10.3** Los campos de bits ayudan a conservar el almacenamiento.

**10.4** Aunque los campos de bits ahorran espacio, utilizarlos puede ocasionar que el compilador genere código en lenguaje máquina de ejecución lenta. Esto ocurre debido a que éste toma operaciones adicionales en lenguaje máquina para acceder sólo a porciones de una unidad de almacenamiento direccionable. Éste es uno de los muchos ejemplos del equilibrio espacio-tiempo que se suscitan en la ciencia de la computación.

## TIPS DE PORTABILIDAD

**10.1** Debido a que el tamaño de los elementos de un tipo en particular depende de la máquina, y debido a que las consideraciones de alineación de almacenamiento también dependen de la máquina, la representación de una estructura también depende de la máquina.

**10.2** Utilice **`typedef`** para ayudar a que un programa sea más portable.

**10.3** Si los datos están almacenados en una unión como un tipo y se hace referencia a ellos como otro tipo, los resultados dependen de la implementación.

**10.4** La cantidad de almacenamiento requerida para almacenar una unión depende de la implementación.

**10.5** Algunas uniones podrían no portarse fácilmente a otros sistemas de cómputo. El que una unión sea portable o no, no siempre depende de los requerimientos de alineación de almacenamiento para los datos miembro de la unión en un sistema dado.

**10.6** Las manipulaciones de datos a nivel de bits dependen de la máquina.

**10.7** El desplazamiento a la derecha es dependiente de la máquina. Aplicar un desplazamiento a la derecha a un entero con signo en algunas máquinas ocasiona que los bits desocupados se llenen con ceros, y en otras que se llenen con unos.

**10.8** Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten campos de bits que crucen los límites de palabras, y otras no lo hacen.

## OBSERVACIÓN DE INGENIERÍA DE SOFTWARE

**10.1** Así como en una definición de **`struct`**, una definición de **`union`** simplemente crea un tipo nuevo. Al colocar una definición de una unión o de una estructura fuera de cualquier función, no se genera una variable global.

## EJERCICIOS DE AUTOEVALUACIÓN

**10.1** Complete los espacios en blanco:

a) Una ____________ es una colección de variables relacionadas bajo el mismo nombre.

b) Una ____________ es una colección de variables bajo el mismo nombre, en el que las variables comparten el mismo almacenamiento.

c) Los bits del resultado de una expresión que utiliza el operador ____________ se establecen en 1, si los bits correspondientes a cada operando se establecen en 1. De lo contrario, los bits se establecen en cero.

d) Las variables declaradas en la definición de una estructura son llamadas sus ____________.

e) Los bits del resultado de una expresión que utiliza el operador ____________ se establecen en 1, si al menos uno de los bits correspondientes a cualquiera de sus operandos se establece en 1. De lo contrario, los bits se establecen en cero.

f) La palabra reservada ____________ introduce la declaración de una estructura.

g) La palabra reservada ____________ se utiliza para crear un sinónimo del tipo de dato previamente definido.

h) Los bits del resultado de una expresión que utiliza el operador ____________ se establecen en 1, si exactamente uno de los bits correspondientes a cualquiera de sus operandos se establece en 1. De lo contrario, los bits se establecen en cero.

i) El operador a nivel de bits AND (**&**) con frecuencia se utiliza para ____________ los bits, lo que sirve para seleccionar ciertos bits mientras otros se hacen cero.
j) La palabra reservada ____________ se utiliza para introducir la definición de una unión.
k) Al nombre de una estructura se le conoce como ____________ de la estructura.
l) Se accede a un miembro de estructura por medio del operador ____________ o ____________.
m) Los operadores ____________ y ____________ se utilizan para desplazar los bits de un valor hacia la izquierda o hacia la derecha, respectivamente.
n) Una ____________ es un conjunto de enteros representados mediante identificadores.

**10.2** Diga si los siguientes enunciados son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.
a) Las estructuras pueden contener variables de un solo tipo de datos.
b) Dos uniones pueden compararse (utilizando **==**) para determinar si son iguales.
c) La etiqueta con el nombre de una estructura es opcional.
d) Los miembros de estructuras diferentes deben tener nombres únicos.
e) La palabra reservada **typedef** se utiliza para definir nuevos tipos de datos.
f) Las estructuras siempre pasan por referencia a las funciones.
g) Las estructuras no deben compararse utilizando los operadores **==** y **!=**.

**10.3** Escriba el código para realizar lo siguiente:
a) Defina una estructura llamada **parte**, que contenga la variable **numeroParte** de tipo **int**, y el arreglo **nombreParte** de tipo **char**, con valores que pueden ser hasta de 25 caracteres (incluyendo al carácter de terminación nulo).
b) Defina a **Parte** para que sea un sinónimo del tipo **struct parte**.
c) Utilice **Parte** para declarar la variable **a** para que sea del tipo **struct parte**, el arreglo **b[ 10 ]** para que sea del tipo **struct parte**, y la variable **ptr** para que sea de tipo apuntador a **struct parte**.
d) Lea un número de parte y un nombre de parte desde el teclado y colóquelos en los miembros individuales de la variable **a**.
e) Asigne los valores miembro de la variable **a** en el elemento 3 del arreglo **b**.
f) Asigne la dirección del arreglo **b** a la variable apuntador **ptr**.
g) Imprima los valores miembro del elemento 3 del arreglo **b**, utilizando la variable **ptr**, y el operador apuntador de la estructura para hacer referencia a los miembros.

**10.4** Encuentre el error en cada uno de los siguientes fragmentos de código:
a) Suponga que **struct carta** se definió para que contuviera dos apuntadores a los tipos **char**, **cara** y **palo**. Además, suponga que la variable **c** se definió para que fuera del tipo **struct carta** y que la variable **ptrC** se definió para que fuera del tipo apuntador a **struct carta**. A la variable **ptrC** se le asignó la dirección de **c**.

```
printf( "%s\n", *ptrC->cara );
```

b) Suponga que **struct carta** se definió para que contuviera dos apuntadores a los tipos **char**, **cara** y **palo**. Además, suponga que el arreglo **corazones[ 13 ]** se definió para que fuera del tipo **struct carta**. La siguiente instrucción debe imprimir el miembro carta del elemento 10 del arreglo.

```
printf( "%s\n", corazones.cara );
```

c)
```
union valores {
   char w;
   float x;
   double y;
};
union valores v = { 1.27 };
```

d)
```
struct persona {
   char apellido[ 15 ];
   char nombre[ 15 ];
   int edad;
}
```

e) Suponga que **struct persona** se definió como en la parte (d), pero con la corrección apropiada.

```
persona d;
```

f) Suponga que la variable **p** se declaró como del tipo **struct persona** y que la variable **c** se declaró como del tipo **struct carta**.

```
p = c;
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**10.1** a) Estructura. b) Unión. c) AND a nivel de bits (**&**). d) Miembros. e) OR incluyente a nivel de bits (**|**). f) **struct**. g) **typedef**. h) OR excluyente a nivel de bits (**^**). i) Enmascarar. j) **union**. k) Etiqueta con el nombre. l) Miembro de la estructura, apuntador de la estructura. m) Operador de desplazamiento a la izquierda (**<<**), operador de desplazamiento a la derecha (**>>**). n) Enumeración.

**10.2** a) Falso. Una estructura puede contener variables de muchos tipos de datos.
b) Falso. Las uniones no pueden compararse debido a que podría haber bites de datos indefinidos con diferentes valores en la unión de variables, los cuales, de otro modo, serían idénticos.
c) Verdadero.
d) Falso. Los miembros de estructuras separadas pueden tener los mismos nombres, pero los miembros de la misma estructura deben tener nombres únicos.
e) Falso. La palabra reservada **typedef** se utiliza para definir nuevos nombres (sinónimos) para tipos de datos previamente definidos.
f) Falso. Las estructuras siempre pasan a funciones mediante una llamada por valor.
g) Verdadero, debido a los problemas de alineación.

**10.3** a)
```
struct parte{
int numeroParte;
char nombreParte[ 25 ];
};
```
b) **typedef struct parte Parte;**
c) **Parte a, b[ 10 ], *ptr;**
d) **scanf( "%d%s", &a.numeroParte, &a.nombreParte );**
e) **b[ 3 ] = a;**
f) **ptr = b;**
g) **printf( "%d %s\n", (ptr + 3 )->numeroParte, (ptr + 3 )->nombreParte);**

**10.4** a) Los paréntesis que deben encerrar ***ptrC** se omitieron, lo que ocasiona que el orden en la evaluación de la expresión sea incorrecto. La expresión debe ser
```
( *ptrC )->cara
```
b) Se omitió el subíndice del arreglo. La expresión debe ser
```
corazones[ 10 ].cara
```
c) Una unión sólo puede inicializarse con un valor que tiene el mismo tipo que el primer miembro de la unión.
d) Es necesario un punto y coma para finalizar la definición de una estructura.
e) La palabra reservada **struct** se omitió en la declaración de la variable. La declaración debe ser
```
struct persona d;
```
f) Las variables de diferentes tipos de estructuras no pueden asignarse entre sí.

## EJERCICIOS

**10.5** Proporcione la definición para cada una de las siguientes estructuras y uniones:
a) La estructura inventario que contiene un arreglo de caracteres **nombreParte[ 30 ]**, la variable entera **numeroParte**, la variable de punto flotante **precio**, y las variables enteras **almacen** y **resurtir**.
b) La unión datos que contiene **char c**, **short s**, **long b**, **float f** y **double d**.
c) Una estructura llamada **direccion** que contiene los arreglos de caracteres **direccionCalle[ 25]**, **ciudad[ 20 ]**, **estado[ 3 ]** y **codigoPostal[ 6 ]**.
d) Una estructura estudiante que contiene los arreglos **nombre[ 15 ]** y **apellido[ 15 ]**, y la variable **direccionCasa** del tipo **struct direccion** de la parte (c).
e) Una estructura prueba que contiene campos de 16 bits con anchos de 1 bit. Los nombres de los campos de bits son las letras de la **a** a la **p**.

**10.6** Dadas las siguientes definiciones de estructuras y variables,
```
struct cliente{
   char apellido[ 15 ];
   char nombre[ 15 ];
   int numeroCliente;
```

```
      struct {
         char numeroTelefonico[ 11 ];
         char direccion[ 50 ];
         char ciudad[ 15 ];
         char estado[ 3 ];
         char codigoPostal[ 6 ];
      } personal;
   } registroCliente, *ptrCliente;
   ptrCliente = &registroCliente;
```

escriba una expresión que pueda utilizarse para acceder a los miembros de la estructura en cada una de las siguientes partes:

a) Al miembro **apellido** de la estructura **registroCliente**.
b) Al miembro **apellido** de la estructura apuntada por **ptrCliente**.
c) Al miembro **nombre** de la estructura **registroCliente**.
d) Al miembro **nombre** de la estructura apuntada por **ptrCliente**.
e) Al miembro **numeroCliente** de la estructura **registroCliente**.
f) Al miembro **numeroCliente** de la estructura apuntada por **ptrCliente**.
g) Al miembro **numeroTelefonico** del miembro **personal** de la estructura **registroCliente**.
h) Al miembro **numeroTelefonico** del miembro **personal** de la estructura apuntada por **ptrCliente**.
i) Al miembro **direccion** del miembro **personal** de la estructura **registroCliente**.
j) Al miembro **direccion** del miembro **personal** de la estructura apuntada por **ptrCliente**.
k) Al miembro **ciudad** del miembro **personal** de la estructura **registroCliente**.
l) Al miembro **ciudad** del miembro **personal** de la estructura apuntada por **ptrCliente**.
m) Al miembro **estado** del miembro **personal** de la estructura **registroCliente**.
n) Al miembro **estado** del miembro **personal** de la estructura apuntada por **ptrCliente**.
o) Al miembro **codigoPostal** del miembro **personal** de la estructura **registroCliente**.
p) Al miembro **codigoPostal** del miembro **personal** de la estructura apuntada por **ptrCliente**.

**10.7** Modifique el programa de la figura 10.16 para barajar las cartas, utilizando un barajado de alto rendimiento (como muestra la figura 10.3). Imprima el mazo resultante en un formato de dos columnas como en la figura 10.4. Preceda a cada carta con su color.

**10.8** Genere una unión entero con miembros **char c**, **short s**, **int i**, y **long b**. Escriba un programa que introduzca un valor de tipo **char**, un **short**, un **int** y un **long**, y que almacene los valores en variables de unión del tipo **union entero**. Cada variable de unión debe imprimirse como un **char**, un **short**, un **int** y un **long**. ¿Los valores siempre se imprimen correctamente?

**10.9** Genere una unión **puntoFlotante** con miembros **float f**, **double d** y **long double x**. Escriba un programa que introduzca un valor de tipo **float**, un **double** y un **long double**, y que almacene los valores en variables de unión del tipo **union puntoFlotante**. Cada variable de unión debe imprimirse como un **float**, un **double** y un **long double**. ¿Los valores siempre se imprimen correctamente?

**10.10** Escriba un programa que desplace hacia la derecha una variable entera de 4 bits. El programa debe imprimir el entero en bits, antes y después de cada operación de desplazamiento. ¿Su sistema coloca **0**s o **1**s en los bits desocupados?

**10.11** Si su computadora utiliza enteros de 2 bytes, modifique el programa de la figura 10.7 para que funcione con enteros de 2 bytes.

**10.12** Desplazar hacia la izquierda un entero **unsigned** un bit es equivalente a multiplicar el valor 2. Escriba una función **potencia2** que tome dos argumentos enteros, **numero** y **potencia**, y que calcule

$$numero * 2^{potencia}$$

Utilice el operador de desplazamiento para calcular el resultado. Imprima los valores como enteros y como bits.

**10.13** El operador de desplazamiento a la izquierda puede utilizarse para empacar dos valores de carácter en una variable entera **unsigned**. Escriba un programa que introduzca dos caracteres desde el teclado y que los pase a la función **empacaCaracteres**. Para empacar dos caracteres en una variable entera **unsigned**, asigne el primer carácter a la variable **unsigned**, desplace 8 posiciones de bits hacia la izquierda la variable **unsigned** y combine la variable **unsigned** con el segundo carácter por medio del operador OR incluyente a nivel de bits. El programa debe arrojar los caracteres en su formato de bits, antes y después de que se empaquen en la variable entera **unsigned**, para demostrar que los caracteres están empacados correctamente en la variable **unsigned**.

**10.14** Por medio del operador de desplazamiento a la derecha, el operador a nivel de bits AND y una máscara, escriba una función **empacaCaracteres** que tome la variable entera **unsigned** del ejercicio 10.13, y que la desempaque en dos caracteres. Para desempacar dos caracteres de una variable entera **unsigned**, combine la variable **unsigned** con la máscara **65280** (**00000000 00000000 11111111 00000000**), y desplace 8 bits hacia la derecha al resultado. Asigne el valor resultante a una variable **char**. Después combine la variable entera **unsigned** con la máscara **255** (**00000000 00000000 00000000 11111111**). Asigne el resultado a otra variable **char**. El programa debe imprimir la variable entera **unsigned** en bits, antes de que sea desempacada, y después debe imprimir los caracteres en bits para confirmar que se desempacaron correctamente.

**10.15** Si su sistema utiliza enteros de 4 bytes, rescriba el programa del ejercicio 10.13 para empacar 4 caracteres.

**10.16** Si su sistema utiliza enteros de 4 bytes, rescriba la función **desempacarCaracteres** del ejercicio 10.14 para desempacar 4 caracteres. Genere las máscaras que necesite para desempacar los cuatro caracteres, desplazando 8 bits hacia la izquierda el valor **255** de la variable **mascara**, 0, 1, 2 o 3 veces (dependiendo del byte que esté desempacando).

**10.17** Escriba un programa que invierta el orden de los bits de un valor entero **unsigned**. El programa debe introducir el valor del usuario y llamar a la función **invierteBits** para imprimir los bits en orden inverso. Imprima el valor en bits, tanto antes como después de que los bits se inviertan, para confirmar que se invirtieron de manera correcta.

**10.18** Modifique la función **despliegaBits** de la figura 10.7 para que sea portable entre sistemas que usen enteros de 2 bytes o enteros de 4 bytes. [*Pista:* Utilice el operador **sizeof** para determinar el tamaño de un entero en una máquina en particular.]

**10.19** El siguiente programa utiliza la función **multiplo** para determinar si el entero introducido desde el teclado es un múltiplo de algún entero **X**. Revise la función **multiplo**, después determine el valor de **X**.

```
/* ej10_19.c */
/* Este programa determina si el valor es un múltiplo de X. */
#include <stdio.h>

int multiplo( int num ); /* prototipo */

int main()
{
   int y; /* y almacenará un entero introducido por el usuario */

   printf( "Introduce un entero entre 1 y 32000: " );
   scanf( "%d", &y );

   /* si y es un múltiplo de X */
   if ( multiplo( y ) ) {
      printf( "%d es un multiplo de X\n", y );
   } /* fin de if */
   else {
      printf( "%d no es un multiplo de X\n", y );
   } /* fin de else */

   return 0; /* indica terminación exitosa */
} /* fin de main */

/* determina si suma es un múltiplo de X */
int multiplo( int num )
{
   int i;            /* contador */
   int mascara = 1; /* inicializa mascara */
   int mult = 1;    /* initialize mult */
```

(Parte 1 de 2.)

```

   for ( i = 1; i <= 10; i++, mascara <<= 1 ) {

      if ( ( num & mascara ) != 0 ) {
         mult = 0;
         break;
      } /* fin de if */

   } /* fin de for */

   return mult;
} /* fin de la función multiplica */
```

(Parte 2 de 2.)

**10.20** ¿Qué es lo que hace el siguiente programa?

```
/* ej10_20.c */
#include <stdio.h>

int misterio( unsigned bits ); /* prototipo */

int main()
{
   unsigned x; /* x almacenará un entero introducido por el usuario  */

   printf( “Introduce un entero: “ );
   scanf( “%u”, &x );

   printf( “El resutado es %d\n”, misterio( x ) );

   return 0; /* indica terminación exitosa */
} /* fin de main */

/* ¿Qué hace la función ? */
int misterio( unsigned bits )
{
   unsigned i;                  /* contador */
   unsigned mascara = 1 << 31; /* inicializa mascara */
   unsigned total = 0;         /* inicializa total */

   for ( i = 1; i <= 32; i++, bits <<= 1 ) {

      if ( ( bits & mascara ) == mascara ) {
         total++;
      } /* fin de if */

   } /* fin de for */

   return !( total % 2 ) ? 1 : 0;
} /* fin de la función misterio */
```

# 11

# Procesamiento de archivos en C

## Objetivos

- Crear, leer, escribir y actualizar archivos.
- Familiarizarse con el procesamiento de archivos de acceso secuencial.
- Familiarizarse con el procesamiento de archivos de acceso aleatorio.




*Leí parte de él en todo el camino hacia allá.*
Samuel Goldwyn

*¡Quítense el sombrero!*
*La bandera está pasando.*
Henry Holcomb Bennett

*La conciencia… no parece dividirse en pequeños bits… parece más natural describirla metafóricamente como un "río" o un "flujo".*
William James

*Solamente puedo suponer que un documento de "No archivar" se coloca en un archivo de "No archivar".*
Senador Frank Church
Subcomité de Inteligencia del Senado, 1975.

## Plan general



## 11.1 Introducción

El almacenamiento de los datos dentro de variables y arreglos es temporal; todos esos datos se pierden cuando termina el programa. Los *archivos* se utilizan para retener permanentemente grandes cantidades de datos. Las computadoras almacenan los archivos en dispositivos secundarios de almacenamiento, en especial en dispositivos de disco. En este capítulo, explicaremos cómo se crean los archivos de datos, cómo se actualizan y cómo se procesan mediante programas en C. Explicaremos los archivos de acceso secuencial y los archivos de acceso aleatorio.

## 11.2 Jerarquía de datos

En última instancia, todos los elementos de datos que procesa la computadora se reducen a combinaciones de *ceros* y *unos*. Esto ocurre debido a que es más sencillo y económico construir dispositivos electrónicos que puedan asumir dos estados estables; uno de los estados representa un **0** y el otro representa un **1**. Es sorprendente que las funciones más impresionantes que realizan las computadoras involucran solamente las manipulaciones fundamentales de **0**s y **1**s.

El elemento de dato más pequeño en una computadora puede asumir el valor **0**, o el valor **1**. A tal elemento de dato se le llama *bit* (abreviatura de "dígito binario"; un dígito binario puede asumir uno de dos valores). Los circuitos de la computadora realizan distintas manipulaciones simples de bits, tales como determinar el valor de un bit, establecer su valor y revertirlo (de **1** a **0** y de **0** a **1**).

Para los programadores es muy difícil trabajar con datos a nivel de bits (una forma de bajo nivel). En lugar de eso, los programadores prefieren trabajar con datos de la forma de *enteros decimales* (es decir, 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), *letras* (es decir, A-Z y a-z), y *símbolos especiales* (es decir, $, @, %, &, *, (, ), -, +, ", :, ?, /, y otros). A los dígitos, las letras y los símbolos especiales se les conoce como *caracteres*. Al conjunto de todos los caracteres que pueden utilizarse para escribir programas y representar elementos de datos en una computadora en particular se le llama *conjunto de caracteres* de la computadora. Dado que las computadoras procesan solamente **1**s y **0**s, cada carácter del conjunto de caracteres de una computadora se representa como un patrón de **1**s y **0**s (llamado *byte*). Actualmente, los bytes están compuestos por ocho bits. Los programadores crean programas y elementos de datos como caracteres; las computadoras manipulan y procesan estos caracteres como patrones de bits.

Así como los caracteres están compuestos por bits, los *campos* están compuestos por caracteres. Un campo es un grupo de caracteres que en forma conjunta representan un mismo significado. Por ejemplo, un campo que consta sólo de letras mayúsculas y minúsculas puede utilizarse para representar el nombre de una persona.

Los elementos de datos que procesan las computadoras forman una *jerarquía de datos* en la cual, los elementos de datos se hacen más grandes y más complejos en su estructura, conforme progresan de bits a caracteres (bytes), de caracteres a campos, y así sucesivamente.

Un *registro* (es decir, una **`estructura`** en **`C`**) está compuesto por varios campos. Por ejemplo, en un sistema de nómina, el registro de un empleado en particular podría estar compuesto por los siguientes campos:

1. Número de seguro social (campo alfanumérico).
2. Nombre (campo alfabético).
3. Dirección (campo alfanumérico).
4. Sueldo por hora (campo numérico).
5. Deducciones (campo numérico).
6. Percepciones de un año a la fecha (campo numérico).
7. Monto por concepto de impuestos (campo numérico).

Así, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada uno de los campos pertenece al mismo empleado. Por supuesto, una empresa en particular puede tener muchos empleados y tener un registro de nómina por cada empleado. Un *archivo* es un grupo de registros relacionados. Por lo general, el archivo de nómina de una empresa contiene un registro para cada empleado. Así, el archivo de nómina para una pequeña empresa podría contener solamente 22 registros, mientras que el archivo de nómina para una empresa grande podría contener 100,000 registros. No es poco común que una empresa tenga cientos, incluso miles de archivos, en donde algunos de ellos contienen miles de millones de caracteres de información. La figura 11.1 ilustra la jerarquía de datos.

Para facilitar la recuperación de registros específicos de un archivo, se elige al menos un campo como *clave de registro*. Por ejemplo, en el registro de nómina descrito en esta sección, por lo general se elegiría como clave de registro el número de Seguro Social.

Existen muchas formas de organizar los registros en un archivo. El tipo de organización de archivos más popular se llama *archivo secuencial*, en donde por lo general los registros se almacenan ordenadamente, de acuerdo con la clave de registro. En el archivo de nómina, por lo general los registros se colocan en orden de acuerdo con el número de Seguro Social. El primer registro de empleado en el archivo contiene el número menor de Seguro Social, y los registros subsiguientes contienen números de Seguro Social cada vez mayores.

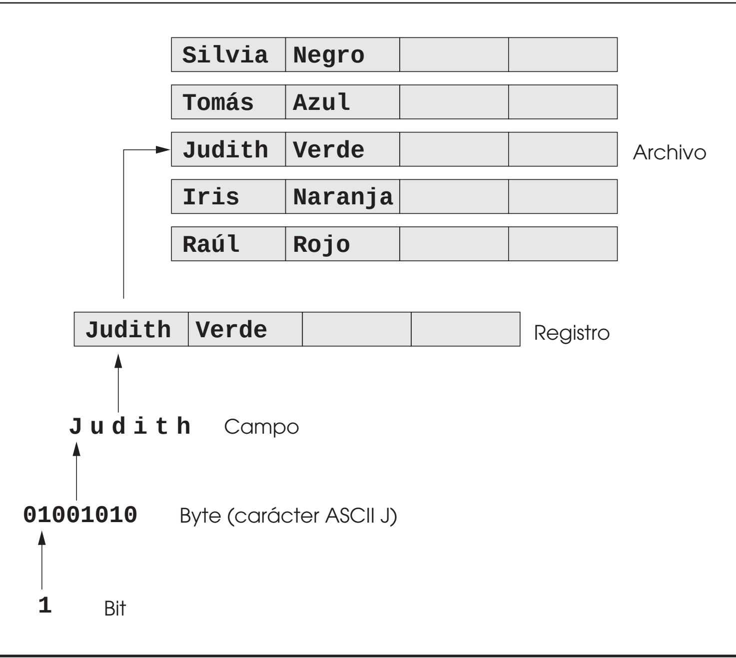



**Figura 11.1** Jerarquía de datos.

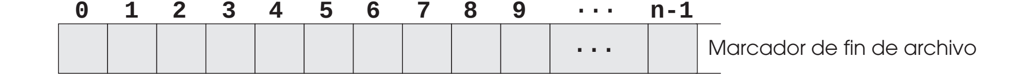



**Figura 11.2** Vista de un archivo de *n* bytes en C.

La mayoría de las empresas almacenan datos en muchos archivos diferentes. Por ejemplo, podrían tener archivos de nómina, archivos de cuentas por cobrar (listan las deudas de dinero de los clientes), cuentas por pagar (listan el dinero que se le debe a los proveedores), archivos de inventario (listan todos los elementos que maneja la empresa) y muchos otros tipos de archivos. En ocasiones, a un grupo de archivos relacionados se le llama base de datos. A una colección de programas diseñados para crear y administrar bases de datos se le llama *sistema de administración de bases de datos* (DBMS).

## 11.3 Archivos y flujos

C ve a cada archivo simplemente como un flujo secuencial de bytes (figura 11.2). Cada archivo termina con una *marca de fin de archivo* o con un número de byte específico almacenado dentro de una estructura de dato administrativa mantenida por el sistema. Cuando se *abre* un archivo, se le asocia un *flujo*. Cuando comienza la ejecución de un programa, se abren tres archivos asociados y sus flujos asociados (de *entrada estándar*, de *salida estándar*, y *de error estándar*). Los flujos proporcionan canales de comunicación entre los archivos y los programas. Por ejemplo, el flujo estándar de entrada permite a un programa leer datos desde el teclado, y el flujo estándar de salida permite al programa desplegar los datos en la pantalla. Al abrir un archivo se devuelve un apuntador a la estructura `FILE` (definida en `<stdio.h>`), la cual contiene información utilizada para procesar el archivo. Esta estructura incluye un *descriptor de archivo*, es decir, un índice dentro de un arreglo del sistema operativo llamado *tabla de archivos abiertos*. Cada elemento del arreglo contiene un *bloque de control de archivo* (*FCB*) que utiliza el sistema operativo para administrar un archivo en particular. La entrada estándar, la salida estándar y el error estándar se manipulan por medio de los apuntadores de archivo `stdin`, `stdout` y `stderr`.

La biblioteca estándar proporciona muchas funciones para leer datos desde archivos y para escribir datos en archivos. La función `fgetc`, al igual que `getchar`, lee un carácter desde un archivo. La función `fgetc` recibe como argumento un apuntador `FILE` para el archivo desde el que se lee el carácter. La llamada a `fgetc( stdin )` lee un carácter desde `stdin`, la entrada estándar. Esta llamada es equivalente a la llamada a `getchar()`. La función `fputc`, al igual que `putchar`, escribe un carácter en un archivo. La función `fputc` recibe como argumentos un carácter a escribir y un apuntador para el archivo en el cual se escribirá el carácter. La llamada a la función `fputc( 'a', stdout )` escribe el carácter `'a'` en `stdout`, la salida estándar. Esta llamada es equivalente a `putchar( 'a' )`.

Muchas otras funciones para leer datos desde la entrada estándar y escribir datos hacia la salida estándar tienen funciones similares de procesamiento de archivos. Por ejemplo, las funciones `fgets` y `fputs` pueden utilizarse para leer una línea desde un archivo y para escribir una línea en un archivo, respectivamente. En el capítulo 8, explicamos sus contrapartes, `gets` y `puts`, para leer desde la entrada estándar y para escribir desde la salida estándar. En las siguientes secciones, presentamos los equivalentes para el procesamiento de archivos de `scanf` y `printf`, `fscanf` y `fprintf`. Más adelante en este capítulo, explicaremos las funciones `fread` y `fwrite`.

## 11.4 Creación de un archivo de acceso secuencial

C no impone una estructura a un archivo. Por lo tanto, las ideas de registro de un archivo no existen como parte del lenguaje C. Entonces, el programador debe proporcionar la estructura del archivo para cumplir con los requerimientos de una aplicación en particular. En el siguiente ejemplo, mostraremos cómo el programador puede insertar una estructura de registro dentro de un archivo.

La figura 11.3 crea un archivo de acceso secuencial que puede utilizarse para un sistema de cuentas por cobrar y ayudar a mantener el registro de los montos de las deudas crediticias de sus clientes. Para cada cliente,

```
/* Figura 11.3: fig11_03.c
   Crea un archivo secuencial */
#include <stdio.h>

int main()
{
   int cuenta;         /* número de cuenta */
   char nombre[ 30 ];  /* nombre de cuenta */
   double saldo;       /* saldo de la cuenta */

   FILE *ptrCf;        /* ptrCf = apuntador al archivo clientes.dat */

   /* fopen abre el archivo. Si no es capaz de crear el archivo, sale del
      programa  */
   if ( ( ptrCf = fopen( “clientes.dat”, “w” ) ) == NULL ) {
      printf( “El archivo no pudo abrirse\n” );
   } /* fin de if */
   else {
      printf( “Introduzca la cuenta, el nombre, y el saldo.\n” );
      printf( “Introduzca EOF al final de la entrada.\n” );
      printf( “? “ );
      scanf( “%d%s%lf”, &cuenta, nombre, &saldo );

      /* escribe la cuenta, el nombre y el saldo dentro del archivo con fprintf */
      while ( !feof( stdin ) ) {
         fprintf( ptrCf, “%d %s %.2f\n”, cuenta, nombre, saldo );
         printf( “? “ );
         scanf( “%d%s%lf”, &cuenta, nombre, &saldo );
      } /* fin de while */

      fclose( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Introduzca la cuenta, el nombre, y el saldo.
Introduzca EOF al final de la entrada.
? 100 Sanchez 24.98
? 200 Lopez 345.67
? 300 Blanco 0.00
? 400 Martinez -42.16
? 500 Rico 224.62
? ^Z
```

**Figura 11.3** Creación de un archivo secuencial.

el programa obtiene un número de cuenta, el nombre del cliente y su saldo (es decir, el monto que el cliente debe a la empresa por concepto de los bienes y servicios recibidos en el pasado). Los datos obtenidos de cada cliente constituyen un “registro” del cliente. En esta aplicación, el número de cuenta se utiliza como la clave del registro; el archivo se creará y se actualizará de acuerdo con el orden de los números de cuenta. Este programa asume que el usuario introduce los registros de acuerdo con el orden de los números de cuenta. En un sistema completo de cuentas por cobrar, puede proporcionarse una capacidad de ordenamiento para que el usuario introduzca los registros en cualquier orden. Los registros entonces se ordenarían y se escribirían en el archivo.

Ahora, revisemos este programa. La línea 11

```
FILE *ptrCf;
```

establece que **ptrCf** es un apuntador a una estructura **FILE**. Un programa en C administra cada archivo con una estructura **FILE** diferente. El programador no necesita conocer las especificaciones de la estructura **FILE** para utilizar los archivos. Más adelante veremos de forma precisa cómo es que la estructura **FILE** dirige indirectamente al bloque de control de archivos del sistema operativo (FCB) de un archivo.

Cada archivo abierto debe tener por separado una declaración de tipo **FILE** que se utiliza para hacer referencia al archivo. La línea 14

```
if ( ( ptrCf = fopen( "clientes.dat", "w" ) ) == NULL )
```

nombra al archivo, **"clientes.dat"**, para que el programa lo utilice, y establece una "línea de comunicación" con el archivo. A la estructura **FILE** para el archivo abierto con **fopen** se le asigna el apuntador de archivo **ptrCf**. La función **fopen** toma dos argumentos: un nombre de archivo y un *modo de apertura del archivo*. El modo de apertura **"w"** indica que el archivo se abrirá para *escritura*. Si un archivo no existe y se abre para escritura, **fopen** crea el archivo. Si abre un archivo existente para escritura, el contenido del archivo se descarta sin advertencia alguna. En el programa, la instrucción **if** se utiliza para determinar si el apuntador de archivo **ptrCf** es **NULL** (es decir el archivo no está abierto). Si es **NULL**, el programa imprime un mensaje de error y termina. De lo contrario, el programa procesa la entrada y la escribe en el archivo.



### Error común de programación 11.1

*Abrir un archivo existente para escritura (**"w"**) cuando, de hecho, el usuario desea preservar el contenido, es un error, ya que hacer esto ocasiona que se descarte el contenido del archivo sin advertencia alguna.*



### Error común de programación 11.2

*Olvidar abrir un archivo antes de intentar hacer referencia a él dentro de un programa, es un error lógico.*

El programa indica al usuario que introduzca los distintos campos de cada registro, o que introduzca el fin de archivo cuando la entrada de datos sea completa. La figura 11.4 lista las combinaciones de teclas para introducir la marca de fin de archivo en distintos sistemas de cómputo.

La línea 24

```
while ( !feof( stdin ) )
```

utiliza la función **feof** para determinar si se estableció el *indicador de fin de archivo* para el archivo al que hace referencia **stdin**. El indicador de fin de archivo informa al programa que ya no existen datos para procesar. En la figura 11.3, el indicador de fin de archivo se establece para la entrada estándar cuando el usuario introduce la combinación de teclas para fin de archivo. El argumento de la función **feof** es un apuntador al archivo que se va a evaluar mediante el indicador de fin de archivo (en este caso, **stdin**). La función devuelve un valor diferente de cero (verdadero) cuando se establece el indicador de fin de archivo; de lo contrario, la función devuelve cero. La instrucción **while**, que incluye la llamada a **feof** en este programa, continúa ejecutando el **while** mientras no se establezca el indicador de fin de archivo.

La línea 25

```
fprintf( ptrCf, "%d %s %.2f\n", cuenta, nombre, saldo );
```

| Sistema operativo | Combinación de teclas |
|---|---|
| UNIX | *<entrar><ctrl>d* |
| Windows | *<ctrl>z* |
| Macintosh | *<ctrl>d* |

**Figura 11.4** Combinaciones de teclas para la marca de fin de archivo en distintos sistemas operativos.

escribe los datos en el archivo **`clientes.dat`**. Los datos pueden recuperarse más tarde, mediante un programa diseñado para leer el archivo (vea la sección 11.5). La función **`fprintf`** es equivalente a **`printf`**, excepto que **`fprintf`** también recibe como argumento un apuntador para el archivo en el que se escribirán los datos.



### Error común de programación 11.3

*Utilizar un apuntador de archivo incorrecto para hacer referencia a un archivo, es un error lógico.*



### Tip para prevenir errores 11.1

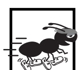
*Asegúrese de que las llamadas a las funciones para procesamiento de archivos dentro de un programa contengan los apuntadores de archivo correctos.*

Después de que el usuario introduce el fin de archivo, el programa cierra el archivo **`clientes.dat`** con **`fclose`** y termina. La función **`fclose`** también recibe como argumento el apuntador de archivo (en lugar del nombre del archivo). Por lo general, si no se llama de manera explícita a la función **`fclose`**, el sistema operativo cerrará el archivo al terminar la ejecución. Éste es un ejemplo de la "limpieza" del sistema operativo.



### Buena práctica de programación 11.1

*Cierre explícitamente cada archivo, en cuanto sepa que el programa no hará referencia a ellos nuevamente.*



### Tip de rendimiento 11.1

*Cerrar un archivo puede liberar recursos para otros usuarios o programas que se encuentren en espera.*

En la ejecución de ejemplo correspondiente al programa de la figura 11.3, el usuario introduce información para cinco cuentas, y después introduce el indicador de fin de archivo para indicar que se completó la entrada de datos. El ejemplo de la ejecución no muestra la forma en que los registros de datos realmente aparecen en el archivo. En la siguiente sección presentaremos un programa que lee el archivo e imprime su contenido, para verificar que el archivo se creó con éxito.

La figura 11.5 ilustra las relaciones entre apuntadores **`FILE`**, estructuras **`FILE`**, y FCBs en memoria. Cuando se abre el archivo **`"clientes.dat"`**, se copia en memoria un FCB para el archivo. La figura muestra la conexión entre el apuntador de archivo devuelto por **`fopen`** y la FCB utilizada por el sistema operativo para administrar el archivo.

Los programas pueden procesar un archivo, varios, o ninguno. Cada archivo utilizado en un programa debe tener un nombre único y tendrá un apuntador de archivo diferente devuelto por **`fopen`**. Una vez que el archivo está abierto, todas las funciones de procesamiento de archivos subsiguientes deben hacer referencia al archivo con el apuntador apropiado. Los archivos pueden abrirse de diferentes maneras (figura 11.6). Para crear un archivo, o para descartar el contenido de un archivo antes de escribir datos, abra el archivo para escritura (**`"w"`**). Para leer un archivo existente, ábralo para lectura (**`"r"`**). Para agregar registros al final de un archivo existente, abra el archivo para agregar (**`"a"`**). Para abrir el archivo de tal manera que pueda escribir en él o leer desde él, abra el archivo para actualización con uno de los siguientes modos, **`"r+"`**, **`"w+"`** o **`"a+"`**. El modo **`"r+"`** abre el archivo para lectura y escritura. El modo **`"w+"`** crea un archivo para lectura y escritura. Si el archivo ya existe, el archivo se abre y el contenido se descarta. El modo **`"a+"`** abre el archivo para lectura y escritura; toda la escritura se hace al final del archivo. Si el archivo no existe, se crea. Observe que cada modo de apertura de archivo tiene un modo binario correspondiente (que contiene la letra **`b`**) para manipular archivos binarios. En las secciones 11.6 a 11.10, cuando introduzcamos los archivos de acceso aleatorio, utilizaremos los modos binarios.

Si ocurre un error mientras se abre un archivo en cualquier modo, **`fopen`** devuelve **`NULL`**.



### Error común de programación 11.4

*Intentar abrir un archivo que no existe, es un error.*



### Error común de programación 11.5

*Intentar abrir un archivo para lectura o escritura sin garantizar los derechos apropiados de acceso al archivo (esto depende del sistema operativo), es un error.*

El usuario tiene acceso a esto

Sólo el sistema operativo tiene acceso a esto

1

`ptrCf = fopen( "clientes.dat", "w" );`

**`fopen`** devuelve un apuntador a la estructura **`FILE`** (definida en **`<stdio.h>`**)

**`ptrNuevo`**

Tabla de Apertura de Archivo

0
1
2
3
4
5
6
7 FCB para **`"clientes.dat"`**
.
.
.

2

La estructura **`FILE`** para **`"clientes.dat"`** contiene un descriptor, es decir, un entero pequeño como índice dentro de la Tabla de Apertura de Archivo

7

4

El programa llama a un servicio del sistema operativo que utiliza los datos de la FCB para controlar toda la entrada y la salida hacia el archivo real en el disco. Nota: el usuario no puede acceder directamente a la FCB

Esta entrada se copia desde la FCB hacia un disco, cuando se abre el archivo

3

Cuando el programa prepara una llamada de E/S tal como

```
fprintf( ptrCf, "%d %s %.2f",
   cuenta, nombre, saldo );
```

el programa localiza el descriptor (7) en la estructura **`FILE`** y lo utiliza para localizar el FCB en la Tabla de Apertura de Archivo

**Figura 11.5** Relación entre los apuntadores **`FILE`**, las estructuras **`FILE`** y las FCBs.

### Error común de programación 11.6

*Intentar abrir un archivo para escritura, cuando no existe espacio disponible, es un error.*



### Error común de programación 11.7

*Intentar abrir un archivo con el modo de apertura incorrecto es un error lógico. Por ejemplo, abrir un archivo con modo de escritura (**`"w"`**) cuando debiera abrirse con modo de actualización (**`"r+"`**) provoca que el contenido del archivo sea descartado.*

| Modo | Descripción |
|---|---|
| `r` | Abre un archivo para lectura. |
| `w` | Crea un archivo para escritura. Si el archivo ya existe, descarta el contenido actual. |
| `a` | Agrega; abre o crea un archivo para escritura al final del archivo. |
| `r+` | Abre un archivo para actualización (lectura y escritura). |
| `w+` | Crea un archivo para actualización. Si el archivo ya existe, descarta el contenido actual. |
| `a+` | Agrega; abre o crea un archivo para actualización; la escritura se hace al final del archivo. |
| `rb` | Abre un archivo para lectura en modo binario. |
| `wb` | Crea un archivo para escritura en modo binario. Si el archivo ya existe, descarta el contenido actual. |
| `ab` | Agrega; abre o crea un archivo para escritura al final del archivo en modo binario. |
| `rb+` | Abre un archivo para actualización (lectura y escritura) en modo binario. |
| `wb+` | Crea un archivo para actualización en modo binario. Si el archivo ya existe, descarta el contenido actual. |
| `ab+` | Agrega; abre o crea un archivo para actualización en modo binario; la escritura se hace al final del archivo. |

**Figura 11.6** Modos de apertura de archivos.



**Tip para prevenir errores 11.2**

*Abra un archivo sólo para lectura (y no para actualización), si el contenido del archivo no debe modificarse. Esto previene modificaciones no intencionales del contenido del archivo. Éste es otro ejemplo del principio del menor privilegio.*

## 11.5 Lectura de datos desde un archivo de acceso secuencial

Los datos se almacenan en archivos para que puedan recuperarse cuando sea necesario procesarlos. En la sección anterior mostramos cómo crear un archivo de acceso secuencial. En esta sección mostramos cómo leer los datos del archivo de manera secuencial.

La figura 11.7 lee los registros desde el archivo **`“clientes.dat”`** creado por el programa de la figura 11.3, e imprime el contenido de dichos registros. La línea 11

```
FILE *ptrCf;
```

indica que **`ptrCf`** es un apuntador a **`FILE`**. La línea 14

```
if ( ( ptrCf = fopen( “clientes.dat”, “r” ) ) == NULL )
```

intenta abrir el archivo **`“clientes.dat”`** para lectura (**`“r”`**), y determina si el archivo se abrió con éxito (es decir, **`fopen`** no devuelve **`NULL`**). La línea 19

```
fscanf( ptrCf, “%d%s%f”, &cuenta, nombre, &saldo );
```

lee un “registro” desde el archivo. La función **`fscanf`** es equivalente a la función **`scanf`**, con la excepción de que **`fscanf`** recibe como argumento el apuntador al archivo desde el que se leen los datos. Después de la

```
1  /* Figura 11.7: fig11_07.c
2     Lectura e impresión de un archivo secuencial */
3  #include <stdio.h>
4
5  int main()
6  {
```

**Figura 11.7** Lectura e impresión de un archivo secuencial. (Parte 1 de 2.)

```
7      int cuenta;         /* número de cuenta */
8      char nombre[ 30 ];  /* nombre de cuenta */
9      double saldo;       /* saldo de la cuenta */
10
11     FILE *ptrCf;        /* ptrCf = apuntador al archivo clientes.dat */
12
13     /* fopen abre el archivo; si no se puede abrir el archivo, abandona el
          programa */
14     if ( ( ptrCf = fopen( "clientes.dat", "r" ) ) == NULL ) {
15        printf( "El archivo no pudo abrirse\n" );
16     } /* fin de if */
17     else { /* lee la cuenta, el nombre y el saldo del archivo */
18        printf( "%-10s%-13s%s\n", "Cuenta", "Nombre", "Saldo" );
19        fscanf( ptrCf, "%d%s%lf", &cuenta, nombre, &saldo );
20
21        /* mientras no sea fin de archivo */
22        while ( !feof( ptrCf ) ) {
23           printf( "%-10d%-13s%7.2f\n", cuenta, nombre, saldo );
24           fscanf( ptrCf, "%d%s%lf", &cuenta, nombre, &saldo );
25        } /* fin de while */
26
27        fclose( ptrCf ); /* fclose cierra el archivo */
28     } /* fin de else */
29
30     return 0; /* indica terminación exitosa */
31
32  } /* fin de main */
```

```
Cuenta   Nombre          Saldo
100      Sanchez         24.98
200      Lopez          345.67
300      Blanco           0.00
400      Martinez       -42.16
500      Rico           224.62
```

**Figura 11.7** Lectura e impresión de un archivo secuencial. (Parte 2 de 2.)

primera ejecución de esta instrucción, **`cuenta`** tendrá el valor **`100`**, **`nombre`** tendrá el valor **`"Sanchez"`** y **`saldo`** tendrá un valor de **`24.98`**. Cada vez que se ejecuta la segunda instrucción **`fscanf`** (línea 24), el programa lee otro registro desde el archivo y, **`cuenta`**, **`nombre`** y **`saldo`** toman nuevos valores. Cuando el programa alcanza el fin de archivo, el archivo se cierra (línea 27) y el programa termina.

Por lo general, para recuperar los datos de un archivo de manera secuencial, un programa inicia la lectura desde el principio del archivo y lee todos los datos consecutivamente hasta que encuentra el dato deseado. Podría ser deseable que los datos se procesaran de manera secuencial varias veces en un archivo (desde el principio del archivo) durante la ejecución de un programa. Una instrucción como

```
rewind( ptrCf );
```

provoca que el *apuntador de posición de archivo* de un programa, el cual indica el número del siguiente byte a leer o a escribir en el archivo, se reubique al principio del archivo (es decir, en el byte 0) al que apunta **`ptrCf`**. El apuntador de posición de archivo en realidad no es un apuntador; es un valor entero que especifica la ubicación del byte en el que ocurrirá la siguiente lectura o escritura dentro del archivo. En ocasiones, a esto se le llama *desplazamiento de archivo*. El apuntador de posición de archivo es un miembro de la estructura **`FILE`** asociada con cada archivo.

El programa de la figura 11.8 permite a un gerente de crédito obtener las listas de los clientes con saldos en cero (es decir, clientes que no deben dinero), clientes con saldos a favor (es decir, clientes a quienes la em-

presa les debe dinero), y clientes con saldos en contra (es decir, clientes que le deben dinero a la empresa por concepto de bienes y servicios recibidos). Un saldo a favor es un monto negativo; un saldo en contra es un monto positivo.

```
/* Figura 11.8: fig11_08.c
   Programa de investigación crediticia */
#include <stdio.h>

/* la función main comienza la ejecución del programa */
int main()
{
   int consulta;       /* número de solicitud */
   int cuenta;         /* número de cuenta */
   double saldo;       /* saldo de la cuenta */
   char nombre[ 30 ];  /* nombre de la cuenta */
   FILE *ptrCf;        /* apuntador al archivo clientes.dat */

   /* fopen abre el archivo; si no se puede abrir el archivo, sale del
      programa */
   if ( ( ptrCf = fopen( “clientes.dat”, “r” ) ) == NULL ) {
      printf( “El archivo no pudo abrirse\n” );
   } /* fin de if */
   else {

      /* despliega las opciones de consulta */
      printf( “Introduzca la consulta\n”
         “ 1 - Lista las cuentas con saldo cero\n”
         “ 2 - Lista las cuentas con saldo a favor\n”
         “ 3 - Lista las cuentas con saldo en contra\n”
         “ 4 - Fin del programa\n? “ );
      scanf( “%d”, &consulta );

      /* procesa la consulta del usuario */
      while ( consulta != 4 ) {

         /* lee la cuenta, el nombre y el saldo del archivo */
         fscanf( ptrCf, “%d%s%lf”, &cuenta, nombre, &saldo );

         switch ( consulta ) {

            case 1:
               printf( “\nCuentas con saldo cero:\n” );

               /* lee el contenido del archivo (hasta eof) */
               while ( !feof( ptrCf ) ) {

                  if ( saldo == 0 ) {
                     printf( “%-10d%-13s%7.2f\n”,
                        cuenta, nombre, saldo );
                  } /* fin de if */

                  /* lee la cuenta, el nombre y el saldo del archivo */
                  fscanf( ptrCf, “%d%s%lf”,
                     &cuenta, nombre, &saldo );
```

**Figura 11.8** Programa de investigación crediticia. (Parte 1 de 2.)

```
               } /* fin de while */

               break;

            case 2:
               printf( “\nCuentas con saldos a favor :\n” );

               /* lee el contenido del archivo (hasta eof) */
               while ( !feof( ptrCf ) ) {

                  if ( saldo < 0 ) {
                     printf( “%-10d%-13s%7.2f\n”,
                        cuenta, nombre, saldo );
                  } /* fin de if */

                  /* lee la cuenta, el nombre y el saldo del archivo */
                  fscanf( ptrCf, “%d%s%lf”,
                     &cuenta, nombre, &saldo );
               } /* fin de while */

               break;

            case 3:
               printf( “\nCuentas con saldo en contra:\n” );

               /* lee el contenido del archivo (hasta eof) */
               while ( !feof( ptrCf ) ) {

                  if ( saldo > 0 ) {
                     printf( “%-10d%-13s%7.2f\n”,
                        cuenta, nombre, saldo );
                  } /* fin de if */

                  /* lee la cuenta, el nombre y el saldo del archivo */
                  fscanf( ptrCf, “%d%s%lf”,
                     &cuenta, nombre, &saldo );
               } /* fin while */

               break;

         } /* fin de switch */

         rewind( ptrCf ); /* devuelve ptrCf al principio del archivo */

         printf( “\n? “ );
         scanf( “%d”, &consulta );
      } /* fin de while */

      printf( “Fin de la ejecucion.\n” );
      fclose( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 11.8** Programa de investigación crediticia. (Parte 2 de 2.)

El programa despliega un menú y permite al gerente de crédito introducir una de tres opciones para obtener información crediticia. La opción 1 produce una lista de cuentas con saldo en cero. La opción 2 produce una lista de cuentas con saldo a favor. La opción 3 produce una lista de cuentas con saldo en contra. La opción 4 termina la ejecución del programa. En la figura 11.9 aparece una muestra de la salida correspondiente a la ejecución del programa.

Observe que los datos en este tipo de archivo secuencial no pueden modificarse sin el riesgo de destruir otros datos del archivo. Por ejemplo, si tuviéramos que cambiar el nombre **"Blanco"** por **"Zubizarreta"**, el nombre viejo no puede simplemente sobrescribirse. El registro para **Blanco** se escribió en el archivo como

```
300 Blanco 0.00
```

Si sobrescribimos el registro comenzando en la misma posición del archivo, utilizando el nuevo nombre, el registro sería

```
300 Zubizarreta 0.00
```

El nuevo registro es mayor (tiene más caracteres) que el registro original. Los caracteres que se encuentran más allá de la **"e"** en **"Zubizarreta"** sobrescribirán el principio del siguiente registro secuencial en el archivo. Aquí, el problema es que los campos (y, por lo tanto, los registros) en el *modelo de entrada/salida con formato* de **fprintf** y **fscanf**, pueden cambiar de tamaño. Por ejemplo, 7, 14, −117, 2024 y 27383 son enteros almacenados internamente en el mismo número de bytes, pero son campos de diferente tamaño cuando se despliegan en la pantalla o se escriben dentro de un archivo de texto.

Por lo tanto, en general, el acceso secuencial con **fprintf** y **fscanf** no se utiliza para actualizar datos en la misma posición. En lugar de eso, usualmente se sobrescribe el archivo completo. Para realizar el cambio de nombre anterior, los registros que se encuentran antes de **300 Blanco 0.00**, en el archivo de acceso secuencial, se copian a un nuevo archivo. Esto requiere procesar cada registro del archivo para actualizar un registro.

```
Introduzca la consulta
 1 - Lista las cuentas con saldo cero
 2 - Lista las cuentas con saldo a favor
 3 - Lista las cuentas con saldo en contra
 4 - Fin del programa
? 1

Cuentas con saldo cero:
300       Blanco          0.00

? 2

Cuentas con saldos a favor :
400       Martinez      -42.16

? 3

Cuentas con saldo en contra:
100       Sanchez        24.98
200       Lopez         345.67
500       Rico          224.62

? 4
Fin de la ejecucion.
```

**Figura 11.9** Salida de ejemplo del programa de investigación crediticia de la figura 11.8.

## 11.6 Archivos de acceso aleatorio

Como mencionamos previamente, los registros de un archivo creados mediante la función de salida con formato **`fprintf`** no necesariamente son de la misma longitud. Sin embargo, los registros individuales de un *archivo de acceso aleatorio* son, por lo general, de la misma longitud y se puede acceder a ellos de modo directo (y por lo tanto, más rápido), sin buscar en otros registros. Esto hace a los archivos de acceso aleatorio apropiados para sistemas de reservaciones en líneas aéreas, sistemas bancarios, sistemas de punto de venta, y otras clases de sistemas de *procesamiento de información* que requieren acceso rápido a un dato específico. Existen otras maneras de implementar los archivos de acceso aleatorio, pero limitaremos nuestra explicación a este método directo, utilizando registros de longitud fija.

Debido a que por lo general cada registro de un archivo de acceso aleatorio tiene la misma longitud, es posible calcular la ubicación exacta de un registro con respecto al principio del archivo, como una función de la clave de registro. Pronto veremos cómo es que esto facilita el acceso inmediato a registros específicos, incluso en archivos grandes.

La figura 11.10 muestra una manera de implementar un archivo de acceso aleatorio. Tal archivo es como un tren de carga con varios vagones; algunos vacíos y otros con carga. Cada vagón del tren tiene la misma longitud.

Los datos pueden insertarse en el archivo de acceso aleatorio, sin destruir otros datos del archivo. Además, los datos previamente almacenados pueden actualizarse o eliminarse sin rescribir el archivo completo. En las siguientes secciones explicaremos cómo realizar cada una de las siguientes tareas en un archivo de acceso aleatorio: introducir datos, leer los datos tanto en modo secuencial como aleatorio, actualizar los datos, y eliminar los datos que ya no necesitamos.

## 11.7 Creación de un archivo de acceso aleatorio

La función **`fwrite`** transfiere un número específico de bytes que comienzan en una ubicación específica de la memoria hacia un archivo. Los datos se escriben al principio de la ubicación del archivo indicada por la posición del apuntador. La función **`fread`** transfiere un número específico de bytes desde la ubicación especificada en el archivo por medio del apuntador de posición del archivo, hacia un área en memoria que comienza con la dirección especificada. Ahora, cuando se escribe un entero, en lugar de utilizar

```
fprintf( ptrF, “%d”, numero );
```

lo que podría imprimir un solo dígito o hasta 11 (10 dígitos más un signo, cada uno de los cuales requiere 1 byte de almacenamiento) para un entero de 4 bytes, puede utilizarse

```
fwrite( &numero, sizeof( int ), 1, ptrF );
```

lo que siempre escribe 4 bytes (o 2 bytes en un sistema con enteros de 2 bytes) desde una variable **`numero`** hacia el archivo representado por **`ptrF`** (más adelante explicaremos el argumento **`1`**). Posteriormente, **`fread`** puede utilizarse para leer 4 de estos bytes dentro de la variable entera **`numero`**. Aunque **`fread`** y **`fwrite`** leen y escriben datos, tales como enteros, de tamaño fijo en lugar de formatos de tamaño variable, los datos que manipulan se procesan en el formato fuente la computadora (es decir, bytes o datos), en lugar del formato de **`printf`** y **`scanf`** legible para el humano.

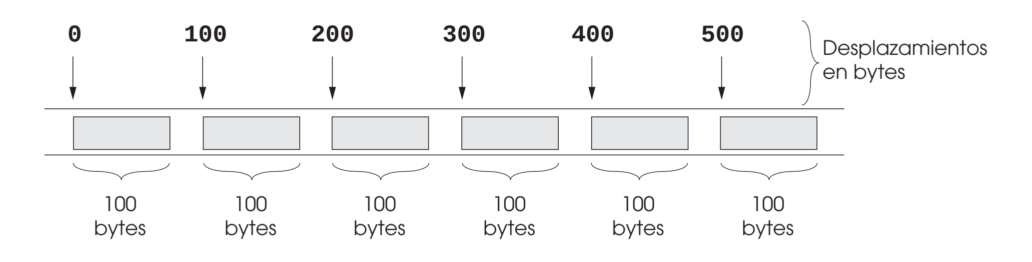



**Figura 11.10** Vista en C de un archivo de acceso aleatorio.

Las funciones **fwrite** y **fread** son capaces de leer y escribir arreglos de datos desde y hacia un disco. El tercer argumento tanto de **fread** como de **fwrite** es el número de elementos del arreglo que debe leerse desde el disco o escribirse en el disco. La llamada anterior a la función **fwrite** escribe un solo entero en disco, así que el tercer argumento es **1** (como si escribiera uno de los elementos del arreglo).

Los programas de procesamiento de archivos rara vez escriben un solo campo a un archivo. Por lo general, escriben una estructura a la vez, como lo muestran los siguientes ejemplos.

Consideremos el siguiente problema:

*Elabore un sistema de procesamiento de créditos capaz de almacenar hasta 100 registros de longitud fija. Cada registro debe consistir en un número de cuenta que se utilizará como clave de registro, un apellido, un nombre y un saldo. El programa resultante debe poder actualizar una cuenta, insertar un nuevo registro de cuenta, eliminar una cuenta y listar todos los registros de cuentas en un archivo de texto con formato para impresión. Utilice un archivo de acceso aleatorio.*

En las siguientes secciones presentaremos las técnicas necesarias para crear el programa de procesamiento de créditos. La figura 11.11 muestra cómo abrir un archivo de acceso aleatorio, cómo definir un formato de

```
/* Figura 11.11: fig11_11.c
   Creación secuencial de un archivo de acceso aleatorio*/
#include <stdio.h>

/* definición de la estructura datosCliente */
struct datosCliente {
   int numCta;           /* número de cuenta */
   char apellido[ 15 ];  /* apellido de la cuenta */
   char nombre[ 10 ];    /* nombre de la cuenta */
   double saldo;         /* saldo de la cuenta */
}; /* fin de la estructura datosCliente */

int main()
{
   int i; /* contador utilizado para contar de 1 a 100 */

   /* crea datosCliente con información predeterminada */
   struct datosCliente clienteEnBlanco = { 0, “”, “”, 0.0 };

   FILE *ptrCf; /* apuntador al archivo credito.dat */

   /* fopen abre el archivo; si no se puede abrir, sale del archivo */
   if ( ( ptrCf = fopen( “credito.dat”, “wb” ) ) == NULL ) {
      printf( “No pudo abrirse el archivo.\n” );
   } /* fin de if */
   else {

      /* escribe 100 registros en blanco en el archivo */
      for ( i = 1; i <= 100; i++ ) {
         fwrite( &clienteEnBlanco, sizeof( struct datosCliente ), 1, ptrCf );
      } /* fin de for */

      fclose ( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 11.11** Creación secuencial de un archivo de acceso aleatorio.

registro por medio de **struct**, cómo escribir datos en el disco y cómo cerrar el archivo. Este programa inicializa con estructuras vacías los 100 registros del archivo **“creditos.dat”**, usando la función **fwrite**. Cada estructura vacía contiene **0** para el número de cuenta, **“ ”** (cadena vacía) para el apellido, **“”** para el nombre, y **0.0** para el saldo. El archivo se inicializa de esta manera para crear espacio en el disco en el que se almacenará el archivo y para poder determinar si un registro contiene datos.

La función **fwrite** escribe un bloque (número específico de bytes) de datos en un archivo. En nuestro programa, la línea 30

```
fwrite( &clienteEnBlanco, sizeof( struct datosCliente ), 1, ptrCf );
```

ocasiona que la estructura **clienteEnBlanco** de tamaño **sizeof( struct datosCliente )** se escriba en el archivo al que apunta **ptrCf**. El operador **sizeof** devuelve el tamaño en bytes de su operando entre paréntesis (en este caso **struct datosCliente**). El operador **sizeof** devuelve un entero sin signo y puede utilizarse para determinar el tamaño en bytes de cualquier tipo de dato o expresión. Por ejemplo, **sizeof( int )** puede utilizarse para determinar si un entero se almacena en 2 o en 4 bytes, en una computadora en particular.

La función **fwrite** realmente puede utilizarse para escribir varios elementos de un arreglo de objetos. Para escribir varios elementos de un arreglo, el programador proporciona un apuntador a un arreglo como el primer argumento en la llamada a **fwrite**, y especifica el número de elementos a escribirse como el tercer argumento de la llamada a **fwrite**. En la instrucción anterior, **fwrite** se utilizó para escribir un objeto sencillo, el cual no era un elemento del arreglo. Escribir un objeto individual es equivalente a escribir un elemento de un arreglo, como el **1** en la llamada a **fwrite**.

## 11.8 Escritura aleatoria de datos en un archivo de acceso aleatorio

La figura 11.12 escribe datos en el archivo **“creditos.dat”**. Utiliza la combinación de **fseek** y **fwrite** para almacenar los datos en una ubicación específica del archivo. La función **fseek** establece el apuntador de posición de archivo en una posición específica del archivo, luego, **fwrite** escribe los datos. En la figura 11.13 mostramos el ejemplo de una ejecución.

```
/* Figura 11.12: fig11_12.c
   Escritura en un archivo de acceso aleatorio */
#include <stdio.h>

/* definición de la estructura datosCliente */
struct datosCliente {
   int numCta;           /* número de cuenta */
   char apellido[ 15 ];  /* apellido de la cuenta */
   char nombre[ 10 ];    /* nombre de la cuenta */
   double saldo;         /* saldo de la cuenta */
}; /* fin de la estructura datosCliente  */

int main()
{
   FILE *ptrCf; /* apuntador al archivo credito.dat */

   /* crea datosCliente con información predeterminada */
   struct datosCliente cliente = { 0, “”, “”, 0.0 };

   /* fopen abre el archivo; si no lo puede abrir, sale del archivo */
   if ( ( ptrCf = fopen( “credito.dat”, “rb+” ) ) == NULL ) {
      printf( “El archivo no pudo abrirse.\n” );
   } /* fin de if */
   else {
```

**Figura 11.12** Escritura aleatoria de datos en un archivo de acceso aleatorio. (Parte 1 de 2.)

```

      /* se requiere que el usuario especifique el número de cuenta */
      printf( “Introduzca el numero de cuenta”
         “ ( 1 a 100, 0 para terminar la entrada )\n? “ );
      scanf( “%d”, &cliente.numCta );

      /* el usuario introduce la información, la cual se copia dentro del
         archivo */
      while ( cliente.numCta != 0 ) {

         /* el usuario introduce el apellido, el nombre y el saldo */
         printf( “Introduzca el apellido, el nombre, el saldo\n? “ );

         /* establece los valores para apellido, nombre y saldo del
            registro */
         fscanf( stdin, “%s%s%lf”, cliente.apellido,
            cliente.nombre, &cliente.saldo );

         /* localiza la posición de un registro específico en el archivo */
         fseek( ptrCf, ( cliente.numCta - 1 ) *
            sizeof( struct datosCliente ), SEEK_SET );

         /* escribe en el archivo la información especificada por el
            usuario */
         fwrite( &cliente, sizeof( struct datosCliente ), 1, ptrCf );

         /* permite al usuario introducir otro número de cuenta */
         printf( “Introduzca el numero de cuenta\n? “ );
         scanf( “%d”, &cliente.numCta );
      } /* fin de while */

      fclose( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 11.12** Escritura aleatoria de datos en un archivo de acceso aleatorio. (Parte 2 de 2.)

```
Introduzca el numero de cuenta ( 1 a 100, 0 para terminar la entrada )
? 37
Introduzca el apellido, el nombre, el saldo
? Baez Daniel 0.00
Introduzca el numero de cuenta
? 29
Introduzca el apellido, el nombre, el saldo
? Brito Nancy -24.54
Introduzca el numero de cuenta
? 96
Introduzca el apellido, el nombre, el saldo
? Sanchez Samuel 34.98
```

**Figura 11.13** Ejecución de ejemplo del programa de escritura aleatoria de datos en un archivo de acceso aleatorio. (Parte 1 de 2.)

```
Introduzca el numero de cuenta
? 88
Introduzca el apellido, el nombre, el saldo
? Santos David 258.34
Introduzca el numero de cuenta
? 33
Introduzca el apellido, el nombre, el saldo
? Roberto Fernandez 314.33
Introduzca el numero de cuenta
? 0447
```

**Figura 11.13** Ejecución de ejemplo del programa de escritura aleatoria de datos en un archivo de acceso aleatorio. (Parte 1 de 2.)

Las líneas 42 y 43

```
fseek( ptrCf, ( cliente.numCta – 1 ) *
   sizeof( struct datosCliente ), SEEK_SET );
```

ubican el apuntador de posición de archivo correspondiente al archivo al que hace referencia `ptrCf` en la posición del byte calculada por `( cliente.numCta - 1 ) * sizeof( struct datosCliente )`. Al valor de esta expresión se le llama el *desplazamiento*. Debido a que el número de cuenta está entre 1 y 100, pero las posiciones de los bytes en el archivo comienzan con `0`, se resta 1 al número de cuenta cuando se calcula la ubicación del byte del registro. Así, para el registro 1, el apuntador de posición de archivo se establece en el byte 0 del archivo. La constante simbólica `SEEK_SET` indica que el apuntador de posición de archivo se ubica en una posición relativa al inicio del archivo, multiplicado por el monto del desplazamiento. Como lo indica la instrucción anterior, la búsqueda del número de cuenta 1 en el archivo coloca el apuntador de posición de archivo al principio del archivo, debido a que la ubicación del byte calculado es 0. La figura 11.14 ilustra el apuntador de archivo que hace referencia a la estructura `FILE` en memoria. El apuntador de posición de archivo en este diagrama indica que el siguiente byte a leerse o escribirse está a 5 bytes a partir del principio del archivo.

El prototipo de la función `fseek` es

```
int fseek( FILE *flujo, long int desplazamiento, int en Donde );
```

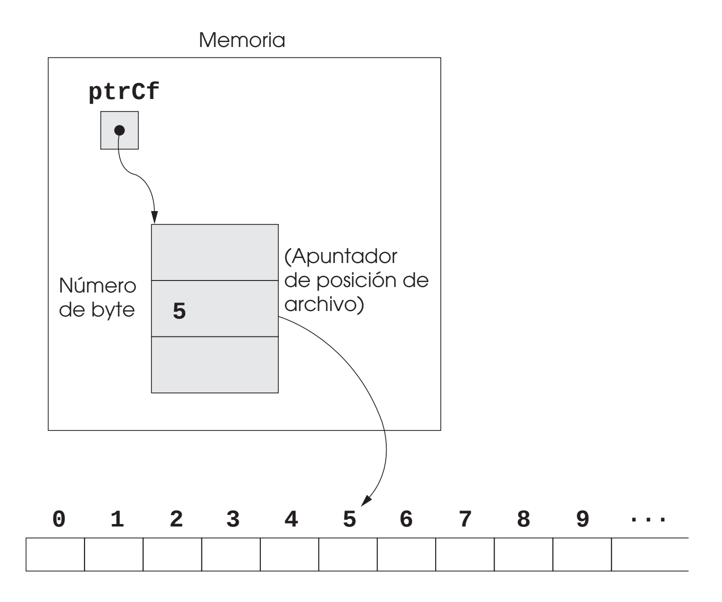



**Figura 11.14** El apuntador de posición de archivo indica un desplazamiento de 5 bytes a partir del principio del archivo.

donde **desplazamiento** es el número de bytes a buscar, desde la ubicación **enDonde** del archivo al que apunta **flujo**. El argumento **enDonde** puede tener uno de tres valores, **SEEK_SET**, **SEEK_CUR** o **SEEK_END** (todos definidos dentro de **<stdio.h>**), los cuales indican la ubicación en el archivo desde donde comienza la búsqueda; **SEEK_SET** indica que la búsqueda comienza al inicio del archivo; **SEEK_CUR** indica que la búsqueda inicia desde la ubicación actual en el archivo; y **SEEK_END** indica que la búsqueda inicia al final del archivo.

## 11.9 Lectura de datos desde un archivo de acceso aleatorio

La función **fread** lee un número específico de bytes de un archivo en memoria. Por ejemplo, la instrucción

```
fread( &cliente, sizeof( struct datosCliente ), 1, ptrCf );
```

lee el número de bytes determinados por **sizeof( struct datosCliente )** desde el archivo al que hace referencia **ptrCf**, y almacena los datos en la estructura **cliente**. Los bytes se leen desde la ubicación especificada por el apuntador de posición dentro del archivo. La función **fread** puede utilizarse para leer varios elementos de tamaño fijo del arreglo, al proporcionar un apuntador al arreglo en el cual se almacenan los elementos y al indicar el número de elementos que pueden leerse. La instrucción anterior especifica que debe leerse un elemento. Para leer más de un elemento especifique el número de elementos en el tercer argumento de la instrucción **fread**.

La figura 11.15 lee de manera secuencial cada registro del archivo **“creditos.dat”**, determina si cada registro contiene datos y despliega los datos con formato correspondientes a los registros que contienen datos. La función **feof** determina cuándo se alcanza el fin de archivo, y la función **fread** transfiere los datos desde el disco hasta la estructura **datosCliente** llamada **cliente**.

```
/* Figura 11.15: fig11_15.c
   Lectura secuencial de un archivo de acceso aleatorio */
#include <stdio.h>

/* definición de la estructura datosCliente */
struct datosCliente {
   int numCta;          /* número de cuenta */
   char apellido[ 15 ]; /* apellido */
   char nombre[ 10 ];   /* nombre */
   double saldo;        /* saldo  */
}; /* fin de la estructura datosCliente */

int main()
{
   FILE *ptrCf; /* apuntador de archivo credito.dat */

   /* crea datosCliente con información predeterminada */
   struct datosCliente cliente = { 0, “”, “”, 0.0 };

   /* fopen abre el archivo; si no se puede abrir, sale del archivo */
   if ( ( ptrCf = fopen( “credito.dat”, “rb” ) ) == NULL ) {
      printf( “No pudo abrirse el archivo.\n” );
   } /* fin de if */
   else {
      printf( “%-6s%-16s%-11s%10s\n”, “Cta”, “Apellido”,
         “Nombre”, “Saldo” );

      /* lee todos los registro del archivo (hasta eof) */
      while ( !feof( ptrCf ) ) {
         fread( &cliente, sizeof( struct datosCliente ), 1, ptrCf );
```

**Figura 11.15** Lectura secuencial de un archivo de acceso aleatorio. (Parte 1 de 2.)

```

         /* despliega el registro */
         if ( cliente.numCta != 0 ) {
            printf( "%-6d%-16s%-11s%10.2f\n",
               cliente.numCta, cliente.apellido,
               cliente.nombre, cliente.saldo );
         } /* fin de if */

      } /* fin de while */

      fclose( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Cta   Apellido         Nombre           Saldo
29    Brito            Nancy           -24.54
33    Fernandez        Roberto         314.33
37    Baez             Daniel            0.00
88    Santos           David           258.34
96    Sanchez          Samuel           34.98
```

**Figura 11.15** Lectura secuencial de un archivo de acceso aleatorio. (Parte 2 de 2.)

## 11.10 Ejemplo práctico: Programa de procesamiento de transacciones

Ahora explicaremos un programa completo de procesamiento de transacciones que utiliza archivos de acceso aleatorio. El programa da mantenimiento a la información de las cuentas de un banco. El programa actualiza las cuentas existentes, agrega cuentas nuevas, elimina las cuentas y almacena una lista de todas las cuentas actuales en un archivo de texto para su impresión. Suponemos que el programa de la figura 11.11 se ejecutó para crear el archivo **`creditos.dat`**.

El programa tiene cinco opciones. La opción 1 llama a la función **`archivoTexto`** para almacenar una lista con formato de todas las cuentas dentro de un archivo de texto llamado **`cuentas.txt`**, el cual podrá imprimirse en cualquier momento. La función utiliza **`fread`** y las técnicas de acceso secuencial utilizadas en el programa de la figura 11.15. Después de elegir la opción 1, el archivo **`cuentas.txt`** contiene:

```
Cta   Apellido         Nombre        Saldo
29    Brito            Nancy        -24.54
33    Fernandez        Roberto      314.33
37    Baez             Daniel         0.00
88    Santos           David        258.34
96    Sanchez          Samuel        34.98
```

La opción 2 llama a la función **`actualizaRegistro`** para actualizar la cuenta. La función solamente actualizará un registro que ya existe, de modo que la función primero verifica si el registro especificado por el usuario está vacío. El registro se lee dentro de la estructura **`cliente`** con **`fread`**, y luego el miembro **`numCuenta`** se compara con **`0`**. Si es igual que **`0`**, el registro no contiene información, y se imprime un mensaje que establece que el registro está vacío. Después, se despliegan las opciones de menú. Si el registro contiene información, la función **`actualizaRegistro`** introduce el monto de la transacción, calcula el nuevo saldo y rescribe el registro en el archivo. Una salida común para la opción 2 es

```
Introduzca cuenta para actualizacion ( 1 - 100 ): 37
37    Baez              Daniel                  0.00

Introduzca el cargo ( + ) o el pago ( - ): +87.99
37    Baez              Daniel                 87.99
```

La opción 3 llama a la función **`registroNuevo`** para agregar una nueva cuenta al archivo. Si el usuario introduce un número de cuenta que ya existe, **`registroNuevo`** despliega un mensaje de error que indica que el registro ya contiene información, y de nuevo se imprimen las opciones del menú. Esta función utiliza el mismo proceso para agregar nuevas cuentas que la figura 11.12. Una salida común para la opción 2 es

```
Introduzca el nuevo numero de cuenta ( 1 - 100 ): 22
Introduzca el apellido, el nombre, y el saldo
? Dominguez Sara 247.45
```

La opción 4 llama a la función **`eliminaRegistro`** para eliminar un registro del archivo. La eliminación se lleva a cabo al solicitar al usuario el número de cuenta y al restablecer el registro. Si la cuenta no contiene información, **`eliminaRegistro`** despliega un mensaje de error que indica que la cuenta no existe. La opción 5 termina la ejecución del programa. La figura 11.16 muestra el programa. Observe que el archivo **`“creditos.dat”`** se abre para actualización (lectura y escritura) mediante el modo **`“rb+”`**.

```
/* Figura 11.16: fig11_16.c
   Este programa lee de manera secuencial un archivo de acceso aleatorio,
   actualiza los datos
   ya escritos en el archivo, crea nuevos datos para colocarlos en el
   archivo, y elimina
   los datos ya existentes en el archivo. */
#include <stdio.h>

/* definición de la estructura datosCliente */
struct datosCliente {
   int numCta;          /* número de cuenta */
   char apellido[ 15 ]; /* apellido */
   char nombre[ 10 ];   /* nombre */
   double saldo;        /* saldo */
}; /* fin de la estructura datosCliente */

/* prototipos */
int intOpcion( void );
void archivoTexto( FILE *ptrLee );
void actualizaRegistro( FILE *ptrF );
void nuevoRegistro( FILE *ptrF );
void eliminaRegistro( FILE *ptrF );

int main()
{
   FILE *ptrCf;   /* apuntador de archivo credito.dat */
   int eleccion;  /* elección del usuario */

   /* fopen abre el archivo; si no se puede abrir, sale del archivo */
   if ( ( ptrCf = fopen( “credito.dat”, “rb+” ) ) == NULL ) {
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 1 de 6.)

```
      printf( "El archivo no pudo abrirse.\n" );
   } /* fin de if */
   else {

      /* permite al usuario especificar una acción */
      while ( ( eleccion = intOpcion() ) != 5 ) {

         switch ( eleccion ) {

            /* crea el archivo desde el registro */
            case 1:
               archivoTexto( ptrCf );
               break;

            /* actualiza registro */
            case 2:
               actualizaRegistro( ptrCf );
               break;

            /* crea registro */
            case 3:
               nuevoRegistro( ptrCf );
               break;

            /* elimina el registro existente */
            case 4:
               eliminaRegistro( ptrCf );
               break;

            /* si el usuario no selecciona una opción válida, despliega un
               mensaje */
            default:
               printf( "Opcion incorrecta\n" );
               break;

         } /* fin de switch */

      } /* fin de while */

      fclose( ptrCf ); /* fclose cierra el archivo */
   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* crea un archivo de texto con formato para impresión */
void archivoTexto( FILE *ptrLee )
{
   FILE *ptrEscribe; /* apuntador del archivo cuentas.txt */

   /* crea datosCliente con información predeterminada */
   struct datosCliente cliente = { 0, "", "", 0.0 };

   /* fopen abre el archivo; si no se puede abrir, sale del archivo */
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 2 de 6.)

```
   if ( ( ptrEscribe = fopen( "cuentas.txt", "w" ) ) == NULL ) {
      printf( "No pudo abrirse el archivo.\n" );
   } /* fin de if */
   else {
      rewind( ptrLee ); /* establece el apuntador en el principio del archivo */
      fprintf( ptrEscribe, "%-6s%-16s%-11s%10s\n",
         "Cta", "Apellido", "Nombre","Saldo" );

      /* copia todos los registros del archivo de acceso aleatorio dentro del
         archivo de texto */
      while ( !feof( ptrLee ) ) {
         fread( &cliente, sizeof( struct datosCliente ), 1, ptrLee );

         /* escribe un registro individual en el archivo de texto */
         if ( cliente.numCta != 0 ) {
            fprintf( ptrEscribe, "%-6d%-16s%-11s%10.2f\n",
               cliente.numCta, cliente.apellido,
               cliente.nombre, cliente.saldo );
         } /* fin de if */

      } /* fin de while */

      fclose( ptrEscribe ); /* fclose cierra el archivo */
   } /* fin de else */

} /* fin de la función archivoTexto */

/* actualiza el saldo en el registro */
void actualizaRegistro( FILE *ptrF )
{
   int cuenta;          /* número de cuenta */
   double transaccion;  /* monto de la transacción */

   /* crea datosCliente sin información  */
   struct datosCliente cliente = { 0, "", "", 0.0 };

   /* obtiene el número de cuenta para actualización */
   printf( "Introduzca cuenta para actualizacion ( 1 - 100 ): " );
   scanf( "%d", &cuenta );

   /* mueve el apuntador de archivo para corregir el registro del archivo */
   fseek( ptrF, ( cuenta - 1 ) * sizeof( struct datosCliente ),
      SEEK_SET );

   /* lee un registro del archivo */
   fread( &cliente, sizeof( struct datosCliente ), 1, ptrF );

   /* despliega un error si la cuenta no existe */
   if ( cliente.numCta == 0 ) {
       printf( "La cuenta #%d no tiene informacion.\n", cuenta );
   } /* fin de if */
   else { /* actualiza el registro */
      printf( "%-6d%-16s%-11s%10.2f\n\n",
         cliente.numCta, cliente.apellido,
         cliente.nombre, cliente.saldo );
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 3 de 6.)

```

      /* pide al usuario el monto de la transacción */
      printf( “Introduzca el cargo ( + ) o el pago ( - ): “ );
      scanf( “%lf”, &transaccion );
      cliente.saldo += transaccion; /* actualiza el saldo del registro */

      printf( “%-6d%-16s%-11s%10.2f\n”,
         cliente.numCta, cliente.apellido,
         cliente.nombre, cliente.saldo );

      /* mueve al apuntador de archivo al registro correcto en el archivo */
      fseek( ptrF, ( cuenta - 1 ) * sizeof( struct datosCliente ),
         SEEK_SET );

      /* escribe el registro actualizado sobre el registro anterior en el
         archivo */
      fwrite( &cliente, sizeof( struct datosCliente ), 1, ptrF );
   } /* fin de else */

} /* fin de la función actualizaRegistro */

/* elimina el registro existente */
void eliminaRegistro( FILE *ptrF )
{

   struct datosCliente cliente; /* almacena los registros leídos en el archivo */
   struct datosCliente clienteEnBlanco = { 0, “”, “”, 0 }; /* cliente en
   blanco */

   int numCuenta; /* número de cuenta */

   /* obtiene el número de cuenta para eliminarlo */
   printf( “Introduzca el numero de cuenta a eliminar ( 1 - 100 ): “ );
   scanf( “%d”, &numCuenta );

   /* mueve el apuntador de archivo al registro correcto en el archivo */
   fseek( ptrF, ( numCuenta - 1 ) * sizeof( struct datosCliente ),
      SEEK_SET );

   /* lee el registro del archivo */
   fread( &cliente, sizeof( struct datosCliente ), 1, ptrF );

   /* si el registro no existe, despliega un error */
   if ( cliente.numCta == 0 ) {
      printf( “La cuenta %d no existe.\n”, numCuenta );
   } /* fin de if */
   else { /* elimina el registro */

      /* mueve el apuntador de archivo hacia el registro correcto en el
         archivo */
      fseek( ptrF, ( numCuenta - 1 ) * sizeof( struct datosCliente ),
         SEEK_SET );

      /* reemplaza el registro existente con un registro en blanco */
      fwrite( &clienteEnBlanco,
         sizeof( struct datosCliente ), 1, ptrF );
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 4 de 6.)

```
   } /* fin de else */

} /* fin de la función eliminaRegistro */

/* crea e inserta un registro */
void nuevoRegistro( FILE *ptrF )
{
   /* crea datosCliente con información predeterminada */
   struct datosCliente cliente = { 0, “”, “”, 0.0 };

   int numCuenta; /* número de cuenta */

   /* obtiene el número de cuenta a crear */
   printf( “Introduzca el nuevo numero de cuenta ( 1 - 100 ): “ );
   scanf( “%d”, &numCuenta );

   /* mueve el apuntador de archivo hacia el registro correcto en el archivo */
   fseek( ptrF, ( numCuenta - 1 ) * sizeof( struct datosCliente ),
      SEEK_SET );

   /* lee el registro desde el archivo */
   fread( &cliente, sizeof( struct datosCliente ), 1, ptrF );

   /* si la cuenta ya existe, despliega un error */
   if ( cliente.numCta != 0 ) {
      printf( “La cuenta #%d ya contiene informacion.\n”,
         cliente.numCta );
   } /* fin de if */
   else { /* crea registro */

      /* el usuario introduce el apellido, el nombre y el saldo */
      printf( “Introduzca el apellido, el nombre, y el saldo\n? “ );
      scanf( “%s%s%lf”, &cliente.apellido, &cliente.nombre,
         &cliente.saldo );

      cliente.numCta = numCuenta;

      /* mueve el apuntador de archivo hacia el registro correcto en el
         archivo */
      fseek( ptrF, ( cliente.numCta - 1 ) *
         sizeof( struct datosCliente ), SEEK_SET );

      /* inserta el registro en el archivo */
      fwrite( &cliente,
         sizeof( struct datosCliente ), 1, ptrF );
   } /* fin de else */

} /* fin de la función nuevoRegistro */

/* inhabilita al usuario para introducir una opción de menú */
int intOpcion( void )
{
   int opcionMenu; /* variable para almacenar la opción del usuario */

   /* despliega las opciones disponibles */
   printf( “\nIntroduzca su opcion\n”
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 5 de 6.)

```
      "1 - almacena un archivo de texto con formato, de las cuentas llamadas\n"
        "    \"cuentas.txt\" para impresion\n"
        "2 - actualiza una cuenta\n"
        "3 - agrega una nueva cuenta\n"
        "4 - elimina una cuenta\n"
        "5 - fin del programa\n? " );

    scanf( "%d", &opcionMenu ); /* recibe la opción del usuario */

    return opcionMenu;

} /* fin de la función introduceOpcion */
```

**Figura 11.16** Programa de cuentas bancarias. (Parte 6 de 6.)

## *RESUMEN*

- Todos los elementos de datos procesados por una computadora se reducen a combinaciones de ceros y unos.
- El elemento de datos más pequeño en una computadora puede asumir el valor de **`0`** o **`1`**. A dicho elemento de dato se le llama bit (la abreviatura de "dígito binario"; un dígito que puede asumir uno de dos valores).
- A los dígitos, a las letras y a los símbolos especiales se les conoce como caracteres. Un conjunto de caracteres es el conjunto de todos los caracteres que pueden utilizarse para escribir programas y representar elementos de datos en una computadora en particular. Cada carácter del conjunto de caracteres de una computadora se representa como un patrón de ocho unos y ceros (llamado byte).
- Un campo es un grupo de caracteres que tienen un significado común.
- Un registro es un grupo de campos relacionados.
- Al menos un campo de cada registro se elige como la clave del registro. La clave de registro identifica a un registro como parte de una persona o entidad en particular.
- El tipo más popular de organización para los registros en un archivo es el llamado archivo de acceso secuencial, en el que se accede a los registros de manera consecutiva hasta que se localiza el dato deseado.
- En ocasiones, a un grupo de campos relacionados se le llama base de datos. A una colección de programas diseñados para crear y manejar una base de datos se le llama sistema de administración de bases de datos (DBMS).
- C ve a un archivo simplemente como un flujo secuencial de bytes.
- C abre tres archivos y sus flujos relacionados (la entrada estándar, la salida estándar y el error estándar) cuando comienza la ejecución de un programa.
- A los apuntadores de archivo asignados a la entrada, a la salida y al error estándar se les llama **`stdin`**, **`stdout`** y **`stderr`**, respectivamente.
- La función **`fgetc`** lee un carácter desde un archivo especificado. La función **`fputc`** escribe un carácter en un archivo especificado.
- La función **`fgets`** lee una línea desde el archivo especificado. La función **`fputs`** escribe una línea en un archivo especificado.
- **`FILE`** es un tipo de estructura definida en el encabezado **`stdio.h`**. El programador no necesita conocer las especificaciones de esta estructura para utilizar archivos. Cuando un archivo se abre, devuelve un apuntador al archivo de la estructura **`FILE`**.
- La función **`fopen`** toma dos argumentos, el nombre del archivo y el modo de apertura, y abre el archivo. Si el archivo existe, su contenido se descarta sin advertencia alguna. Si el archivo no existe y se abre para escritura, **`fopen`** crea el archivo.
- La función **`feof`** determina si se activó el indicador de fin de archivo.
- La función **`fprintf`** es equivalenete a **`printf`**, con la excepción de que **`printf`** recibe como argumento un apuntador al archivo en donde se escribirán los datos.
- La función **`fclose`** cierra el archivo al que apunta su argumento.
- Para crear un archivo o para descartar el contenido de un archivo antes de escribir los datos, abra el archivo para escritura (**`"w"`**). Para leer un archivo existente, abra el archivo para lectura (**`"r"`**). Para agregar registros al final de un archivo existente, abra el archivo para agregar (**`"a"`**). Para abrir un archivo de modo que se pueda escribir o leer en él, abra el ar-

chivo para actualización en alguna de estas tres formas, **"r+"**, **"w+"** o **"a+"**. El modo **"r+"** simplemente abre el archivo para lectura y escritura. El modo **"w+"** crea el archivo si éste no existe y, si existe, descarta el contenido actual del archivo. El modo **"a+"** crea el archivo si no existe, y la escritura se hace al final del archivo.

- La función **fscanf** es equivalente a **scanf**, con la excepción de que **fscanf** recibe como argumento un apuntador al archivo (por lo general, diferente a **stdin**) desde el cual se leerán los datos.
- La función **rewind** provoca que el programa reubique la posición del apuntador de posición del archivo especificado al principio del archivo.
- El procesamiento de archivos de acceso aleatorio se utiliza para acceder directamente a un registro.
- Para facilitar el acceso aleatorio, los datos se almacenan en registros de longitud fija. Dado que cada registro tiene la misma longitud, la computadora puede calcular rápidamente (como una función de la clave de registro) la ubicación exacta de un registro con respecto al principio del archivo.
- Los datos pueden agregarse con facilidad a un archivo de acceso aleatorio sin destruir otros datos del archivo. Los datos almacenados previamente en un archivo con registros de longitud fija también pueden modificarse y eliminarse sin rescribir el archivo completo.
- La función **fwrite** escribe un bloque (un número específico de bytes) de datos en un archivo.
- El operador en tiempo de compilación **sizeof** devuelve el tamaño de su operando.
- La función **fseek** establece el apuntador de posición del archivo en una posición específica en el archivo, basándose en la posición inicial de la búsqueda en el archivo. La búsqueda puede comenzar desde una de tres ubicaciones; **SEEK_SET** comienza desde el principio del archivo, **SEEK_CUR** comienza desde la posición actual del archivo, y **SEEK_END** comienza desde el final del archivo.
- La función **fread** lee un bloque (un número específico de bytes) de datos de un archivo.

## *TERMINOLOGÍA*

**a**, modo de apertura de archivo
**a+**, modo de apertura de archivo
abrir un archivo
abrir una tabla de archivo
apuntador de archivo
apuntador de posición de archivo
archivo
archivo de acceso aleatorio
archivo de acceso secuencial
base de datos
bit
bloque de control de archivo
byte
campo
carácter
ceros y unos
cerrar un archivo
clave de registro
conjunto de caracteres
descriptor de archivo
desplazamiento
desplazamiento de archivo
dígito binario
dígito decimal
entrada/salida con formato
escritura en un archivo
estructura **FILE**
**fclose**
**feof**
**fgetc**
**fgets**
flujo
**fopen**
**fprintf**
**fputc**
**fputs**
**fread**
**fscanf**
**fseek**
**fwrite**
**getchar**
**gets**
jerarquía de datos
marcador de fin de archivo
modo de apertura de un archivo
nombre de archivo
**NULL**
**printf**
procesamiento de transacciones
**putchar**
**puts**
**r**, modo de apertura de archivo
**r+**, modo de apertura de archivo
registro
**rewind**
**scanf**
**SEEK_CUR**
**SEEK_SET**
**SEK_END**
símbolo especial
sistema de administración de base de datos
**stderr** (error estándar)
**stdin** (entrada estándar)
**stdout** (salida estándar)
**w**, modo de apertura de archivo
**w+**, modo de apertura de archivo

## *ERRORES COMUNES DE PROGRAMACIÓN*

**11.1** Abrir un archivo existente para escritura (**"w"**) cuando, de hecho, el usuario desea preservar el contenido, es un error, ya que hacer esto ocasiona que se descarte el contenido del archivo sin advertencia alguna.

**11.2** Olvidar abrir un archivo antes de intentar hacer referencia a él dentro de un programa, es un error lógico.

**11.3** Utilizar un apuntador de archivo incorrecto para hacer referencia a un archivo, es un error lógico.

**11.4** Intentar abrir un archivo que no existe, es un error.

**11.5** Intentar abrir un archivo para lectura o escritura sin garantizar los derechos apropiados de acceso al archivo (esto depende del sistema operativo), es un error.

**11.6** Intentar abrir un archivo para escritura, cuando no existe espacio disponible, es un error.

**11.7** Intentar abrir un archivo con el modo de apertura incorrecto es un error lógico. Por ejemplo, abrir un archivo con modo de escritura (**`"w"`**) cuando debiera abrirse con modo de actualización (**`"r+"`**) provoca que el contenido del archivo sea descartado.

## *TIPS PARA PREVENIR ERRORES*

**11.1** Asegúrese de que las llamadas a las funciones para procesamiento de archivos dentro de un programa contengan los apuntadores de archivo correctos.

**11.2** Abra un archivo sólo para lectura (y no para actualización), si el contenido del archivo no debe modificarse. Esto previene modificaciones no intencionales del contenido del archivo. Éste es otro ejemplo del principio del menor privilegio.

## *BUENA PRÁCTICA DE PROGRAMACIÓN*

**11.1** Cierre explícitamente cada archivo, en cuanto sepa que el programa no hará referencia a ellos nuevamente.

## *TIP DE RENDIMIENTO*

**11.1** Cerrar un archivo puede liberar recursos para otros usuarios o programas que se encuentren en espera.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**11.1** Complete los espacios en blanco:

a) En última instancia, todos los elementos de datos que procesa la computadora se reducen a una combinación de ________ y ________.
b) Al elemento de dato más pequeño que una computadora puede procesar se le llama ________.
c) Un ________ es un grupo de registros relacionados.
d) A los dígitos, letras y símbolos especiales se les denomina ________.
e) A un grupo de archivos relacionados se les llama ________.
f) La función ________ cierra un archivo.
g) La función ________ lee los datos desde un archivo de manera similar a la forma en que **`scanf`** lee desde **`stdin`**.
h) La función ________ lee un carácter desde un archivo especificado.
i) La función ________ lee una línea desde un archivo especificado.
j) La función ________ abre un archivo.
k) Por lo general, la función ________ se utiliza cuando se leen datos desde un archivo en aplicaciones de acceso aleatorio.
l) La función ________ reubica la posición del apuntador de posición de archivo a una ubicación específica en el archivo.

**11.2** Establezca cuál de los siguientes enunciados es *verdadero* y cuál es *falso*. Si es *falso*, explique por qué.

a) La función **`fscanf`** no puede utilizarse para leer datos desde la entrada estándar.
b) El programador debe utilizar explícitamente **`fopen`** para abrir los flujos de entrada estándar, de salida estándar y de error.
c) Un programa debe llamar explícitamente a la función **`fclose`** para cerrar un archivo.
d) Si el apuntador de posición de archivo apunta hacia una ubicación de un archivo secuencial diferente al principio del archivo, el archivo debe cerrarse y reabrirse para leer desde el principio del archivo.
e) La función **`fprintf`** puede escribir en la salida estándar.
f) Los datos de los archivos de acceso secuencial siempre se actualizan sobrescribiendo otros datos.
g) No es necesario buscar a través de todos los registros de un archivo de acceso aleatorio para encontrar un registro específico.

h) Los registros en los archivos de acceso aleatorio no tienen una longitud uniforme.
i) La función **fseek** sólo puede hacer búsquedas relativas al principio del archivo.

**11.3** Escriba una instrucción sencilla para llevar a cabo cada una de las siguientes tareas. Suponga que cada una de estas instrucciones se aplican al mismo programa.

a) Escriba una instrucción que abra el archivo **“maesviej.dat”** para lectura, y asigne el apuntador de archivo devuelto a **ptrF**.
b) Escriba una instrucción que abra el archivo **“trans.dat”** para lectura y que asigne el apuntador de archivo de retorno a **ptrTf**.
c) Escriba una instrucción que abra el archivo **“maesnuev.dat”** para escritura (y creación) y que asigne el apuntador de archivo devuelto a **ptrN**.
d) Escriba una instrucción que lea un registro del archivo **“maesviej.dat”**. El registro consiste en el entero **numCuenta**, la cadena **nombre** y el número de punto flotante **saldoActual**.
e) Escriba una instrucción que lea un registro desde el archivo **“trans.dat”**. El registro consiste en un entero llamado **numCuenta** y el valor de punto flotante **montoMoneda**.
f) Escriba una instrucción que escriba un registro en el archivo **“maesviej.dat”**. El registro consiste en un entero llamado **numCuenta** y el valor de punto flotante **montoMoneda**.

**11.4** Encuentre el error en cada uno de los siguientes segmentos de programa. Explique cómo puede corregirse el error.

a) El archivo al que hace referencia **ptrF(“porPagar.dat”)** no se ha abierto.
```
printf( ptrF, “%d%s%d\n”, cuenta, empresa, monto);
```
b)
```
open( “porCobrar.dat”, “r+”);
```
c) La siguiente instrucción debe leer un registro desde el archivo **“porPagar.dat”**. El apuntador de archivo **ptrPagar** hace referencia a este archivo, y el apuntador de archivo **ptrCobrar** hace referencia a **“porCobrar.dat”**:
```
scanf( ptrCobrar, “%d%s%d\n”, &cuenta, empresa, &monto );
```
d) El archivo **“utilidad.dat”** debe abrirse para agregar datos en él, sin descartar los datos actuales.
```
If ( ( ptrTf = fopen( “utilidad.dat”, “w” ) ) != NULL )
```
e) El archivo **“cursos.dat”** debe abrirse para agregar datos, sin modificar el contenido actual del archivo.
```
if( ( ptrCf = fopen( “cursos.dat”, “w+” ) ) != NULL )
```

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**11.1** a) ls, 0s. b) Bit. c) Archivo. d) Caracteres. e) Base de datos. f) **fclose**. g) **fscanf**. h) **fgetc**. i) **fgets**. j) **fopen**. k) **fread**. l) **fseek**.

**11.2** a) Falso. La función **fscanf** puede utilizarse desde la entrada estándar, incluyendo un apuntador al flujo estándar de entrada.
b) Falso. El compilador de C abre de manera automática estos tres flujos, cuando comienza la ejecución del programa.
c) Falso. Los archivos se cierran cuando termina la ejecución del programa, pero todos los archivos deben cerrarse explícitamente con **fclose**.
d) La función **rewind** puede utilizarse para reubicar el apuntador de posición al principio del archivo.
e) Verdadero.
f) Falso. En la mayoría de los casos, los registros de un archivo no tienen una longitud uniforme. Por lo tanto, es posible que la actualización de un registro provoque la sobrescritura de otros datos.
g) Verdadero.
h) Falso. Los registros de un archivo de acceso aleatorio por lo general tienen una longitud uniforme.
i) Falso. Es posible buscar desde el principio del archivo, desde el final del archivo y desde la posición actual del archivo.

**11.3**
a) `ptrOf = fopen( “viej.dat”, “r” );`
b) `ptrTf = fopen( “trans.dat, “r” );`
c) `ptrNf = fopen( “nuev.dat”, “w” );`
d) `fscanf ( ptrOf, “%d%s%f”, &numCuenta, nombre, &saldoActual );`
e) `fscanf ( ptrTf, “%d%f”, &numCuenta, &montoMoneda );`
f) `fprintf( ptrNf, “%d %s %.2f”, numCuenta, nombre, saldoActual);`

**11.4** a) Error: el archivo **“porPagar.dat”** no se abrió antes de la referencia a su apuntador de archivo.
Corrección: utilice **fopen** para abrir **“porPagar.dat”** para escribir, agregar o actualizar.
b) Error: la función **open** no es una función de ANSI C.
Corrección: utilice la función **fopen**.

c) Error: la función **fscanf** utiliza el apuntador de archivo incorrecto para hacer referencia al archivo **"porPagar.dat"**.
Corrección: utilice el apuntador de archivo **ptrPagar** para hacer referencia a **"porPagar.dat"**.
d) Error: el contenido del archivo se descarta debido a que el archivo se abrió para escritura (**"w"**). Corrección: para agregar datos a un archivo, abra el archivo para actualización (**"r+"**) o abra el archivo para agregar (**"a"**).
e) Error: el archivo **"cursos.dat"** se abrió para actualización con **"w+"**, el cual descarta el contenido actual del archivo.
Corrección: abra el archivo en modo **"a"**.

## *EJERCICIOS*

**11.5** Complete los espacios en blanco:
a) Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundario como ____________.
b) Un ____________ está compuesto por varios campos.
c) A un campo que puede contener dígitos, letras, y espacios en blanco se le llama campo ____________.
d) Para facilitar la recuperación de registros específicos de un archivo, se elige un campo en cada registro como ____________.
e) La gran mayoría de la información que se almacena en la computadora se almacena en archivos ____________.
f) A un grupo de caracteres relacionados y que tienen un significado común se le llama ____________.
g) A los apuntadores de archivos para los tres archivos que se abren de manera automática cuando comienza la ejecución de un programa se les llama ____________, ____________, ____________.
h) La función ____________ escribe un carácter en el archivo especificado.
i) La función ____________ escribe una línea en el archivo especificado.
j) Por lo general, la función ____________ se utiliza para escribir datos en un archivo de acceso aleatorio.
k) La función ____________ reubica al apuntador de posición de archivo al principio del archivo.

**11.6** Establezca si las siguientes frases son *verdaderas* o *falsas*. Si es *falsa*, explique por qué.
a) Las impresionantes funciones que realizan las computadoras involucran esencialmente la manipulación de ceros y unos.
b) La gente prefiere manipular bits en lugar de caracteres y campos, debido a que los bits son más compactos.
c) La gente especifica programas y elementos de datos como caracteres; después, las computadoras manipulan y procesan estos caracteres como grupos de ceros y unos.
d) El código postal de una persona es un ejemplo de un campo numérico.
e) Por lo general, la dirección de la calle de una persona se considera un campo alfabético en las aplicaciones de las computadoras.
f) Los elementos de datos procesados por la computadora forman una jerarquía de datos en la que los elementos se vuelven más complejos, conforme progresan de campos a caracteres, de caracteres a bits, etcétera.
g) Una clave de registro identifica un registro que pertenece a un campo en particular.
h) La mayoría de las empresas almacenan toda su información en un solo archivo, para facilitar su procesamiento en la computadora.
i) En los programas en C, siempre se hace referencia a los archivos por medio de su nombre.
j) Cuando un programa crea un archivo, la computadora lo retiene automáticamente para futuras referencias.

**11.7** El ejercicio 11.3 pide al lector que escriba una serie de instrucciones sencillas. En realidad, estas instrucciones forman el núcleo de un importante tipo de programa de procesamiento de archivos, a saber, un programa de coincidencia de archivos. En el procesamiento de datos comerciales, es común tener varios archivos en cada sistema. Por ejemplo, en un sistema de cuentas por cobrar, por lo general existe un archivo maestro que contiene la información detallada sobre cada cliente, tal como su nombre, su dirección, su número telefónico, su saldo, el límite de crédito, los términos de descuento, las condiciones del contrato y posiblemente una historia condensada de las compras recientes y de los pagos en efectivo.

Conforme ocurren las transacciones (es decir, las ventas hechas y los pagos en efectivo llegan en el correo), éstas se almacenan dentro de un archivo. Al final de cada periodo comercial (es decir, un mes para algunas empresas, una semana para algunas otras y un día en otros casos), el archivo de transacciones (llamado **"trans.dat"** en el ejercicio 11.3) se aplica al archivo maestro (llamado **"maesviej.dat"** en el ejercicio 11.3), y de este modo actualiza cada registro de cuenta con las compras y los pagos. Después de la ejecución de cada una de estas actualizaciones, el archivo maestro se sobrescribe como un archivo nuevo (**"maesnuev.dat"**), el cual se utiliza para el siguiente periodo comercial y para comenzar de nuevo el proceso de actualización.

Los programas de coincidencia de archivos deben lidiar con ciertos problemas que no existen en los programas de un solo archivo. Por ejemplo, no siempre ocurre una coincidencia. Un cliente en el archivo maestro podría no hacer compras o pagos en efectivo durante el periodo comercial actual y, por lo tanto, no aparecerá registro alguno de este cliente en el archivo de transacciones. De modo similar, un cliente que hizo algunas compras o pagos en efectivo podría haberse mudado a esta localidad, y la empresa no tuvo la oportunidad de crear un registro maestro para este cliente.

Utilice las instrucciones escritas en el ejercicio 11.3 como base para escribir un programa completo de coincidencia de cuentas por cobrar. Utilice el número de cuenta de cada archivo como la clave del registro, para efectos de coincidencia. Suponga que cada archivo es un archivo secuencial con los registros almacenados en orden creciente de número de cuenta.

Cuando ocurra una coincidencia (es decir, los registros de la misma cuenta aparecen tanto en el archivo maestro como en el archivo de transacciones), agregue el monto en moneda del archivo de transacciones al saldo actual del archivo maestro y escriba el registro en **"maesnuev.dat"**. (Suponga que la compra se indica con montos positivos en el archivo de transacciones, y que los pagos se indican con montos negativos.) Cuando exista un registro maestro para una cuenta en particular, pero no un registro de transacción correspondiente, solamente escriba el registro maestro dentro de **"maesnuev.dat"**. Cuando exista un registro de transacción, pero no un registro maestro correspondiente, imprima el mensaje **"El registro de transacción no coincide con el número de cuenta … "** (llene el número de cuenta a partir del registro de la transacción).

**11.8** Después de escribir el programa del ejercicio 11.7, escriba un programa para crear algunos datos de prueba que evalúe el programa del ejercicio 11.7. Utilice el siguiente ejemplo de datos de cuentas:

| Archivo Maestro:<br>Número de cuenta | Nombre | Saldo |
|---|---|---|
| **100** | **Alejandro Pérez** | **348.17** |
| **300** | **María Sánchez** | **27.19** |
| **500** | **Samuel Jiménez** | **0.00** |
| **700** | **Susana Salcedo** | **-14.22** |

| Archivo de Transacciones:<br>Número de cuenta | Monto |
|---|---|
| **100** | **27.14** |
| **300** | **62.11** |
| **400** | **100.56** |
| **900** | **82.17** |

**11.9** Ejecute el programa del ejercicio 11.7, utilizando los datos de prueba creados en el ejercicio 11.8. Utilice el programa listado en la sección 11.7 para imprimir el nuevo archivo maestro. Verifique cuidadosamente los resultados.

**11.10** Es posible (en realidad común) tener varios registros de transacciones con la misma clave de registro. Esto ocurre debido a que un cliente en particular pudo haber hecho varias compras y pagos en efectivo durante un periodo comercial. Rescriba su programa de cuentas por cobrar del ejercicio 11.7 para proporcionar la posibilidad de manejar varios registros de transacción con la misma clave de registro. Modifique los datos de prueba del ejercicio 11.8 para incluir los siguientes registros de transacciones adicionales:

| Número de cuenta | Monto |
|---|---|
| **300** | **83.89** |
| **700** | **80.78** |
| **700** | **1.53** |

**11.11** Escriba instrucciones que realicen cada una de las siguientes tareas. Suponga que se definió la estructura

```
struct persona {
 char apellido[ 15 ];
 char nombre[ 15 ];
 char edad[ 4 ];
};
```

y que el archivo ya está abierto para escritura.

a) Inicialice el archivo **“nomedad.dat”** de manera que existan 100 registros con **apellido = “sin-asignar”, nombre= “”** y **edad=“0”**.
b) Introduzca 10 apellidos, nombres y edades, y escríbalos en el archivo.
c) Actualice un registro; si no existe información en el registro, indique al usuario que **“No hay información”**.
d) Elimine un registro que tenga información, por medio de la reinicialización de dicho registro en particular.

**11.12** Usted es el dueño de una tienda de herramientas y necesita mantener un inventario que le pueda decir cuáles herramientas tiene, cuántas tiene y el costo de cada una. Escriba un programa que inicialice el archivo **“herram.dat”** con 100 registros vacíos, que le permita introducir los datos relacionados con cada herramienta, que le permita listar todas sus herramientas, que le permita eliminar un registro de una herramienta que ya no tiene, y que le permita actualizar *cualquier* información en el archivo. El número de identificación de cada herramienta debe ser su número de registro. Utilice la siguiente información para comenzar su archivo:

| # Registro | Nombre de la Herramienta | Cantidad | Costo |
|---|---|---|---|
| 3 | Lijadora eléctrica | 7 | 57.98 |
| 17 | Martillo | 76 | 11.99 |
| 24 | Guía de serrucho | 21 | 11.00 |
| 39 | Podadora | 3 | 79.50 |
| 56 | Sierra mecánica | 18 | 99.99 |
| 68 | Destornillador | 106 | 6.99 |
| 77 | Mazo | 11 | 21.50 |
| 83 | Llave inglesa | 34 | 7.50 |

**11.13** *Generador de números telefónicos con palabras.* Los números telefónicos estándares contienen dígitos de 0 a 9. Los números 2 a 9 contienen, cada uno, tres letras asociadas, como indica la siguiente tabla:

| Dígito | Letra |
|---|---|
| 2 | A B C |
| 3 | D E F |
| 4 | G H I |
| 5 | J K L |
| 6 | M N O |
| 7 | P R S |
| 8 | T U V |
| 9 | W X Z |

A mucha gente le parece difícil memorizar los números telefónicos, de modo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico es 686-2377 podría utilizar la correspondencia que indica la tabla anterior para desarrollar la palabra de siete letras **“NUMEROS”**.

Con frecuencia, las empresas intentan obtener números telefónicos que sean fáciles de recordar por sus clientes. Si una empresa puede anunciar una palabra simple para que los clientes la marquen, entonces sin duda alguna, la empresa recibirá unas cuantas llamadas más.

Cada palabra de siete letras corresponde exactamente a un número telefónico de siete números. El restaurante que desea incrementar sus pedidos a domicilio seguramente podría hacer eso con el número 553-8356 (es decir, **"LLEVELO"**).

Cada número telefónico de siete dígitos corresponde a muchas palabras de siete letras. Desafortunadamente, la mayoría de éstas representan solamente yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una barbería se sintiera contento de saber que el número telefónico de su negocio, 222-3556, corresponde a **"CABELLO"**. El dueño de una tienda de licores sin duda estaría encantado de saber que el teléfono de su negocio, 542-6737, corresponde a **"LICORES"**. Un veterinario cuyo teléfono fuera 627-2682, estaría encantado de que corresponde a **"MASCOTA"**.

Escriba un programa en C que, dado un número de siete dígitos, escriba en un archivo cada posible palabra de siete letras que corresponda al número. Existen 2187 (3 a la séptima potencia) de tales palabras. Evite los números telefónicos con los dígitos 0 y 1.

**11.14** Si usted tiene un diccionario computarizado, modifique el programa que escribió en el ejercicio 11.13 para buscar las palabras en el diccionario. Algunas combinaciones de siete letras creadas por este programa consisten en dos o más palabras (el número telefónico 333-4337 produce **"ELLIDER"**).

**11.15** Modifique el ejemplo de la figura 8.14 para utilizar las funciones **fgetc** y **fputs**, en lugar de **getchar** y **puts**. El programa debe dar al usuario la opción de leer desde la entrada estándar, y de escribir en la salida estándar, o de leer desde un archivo específico y de escribir en un archivo específico. Si el usuario elige la segunda opción, haga que el usuario introduzca los nombres del archivo para la entrada y para la salida.

**11.16** Escriba un programa que utilice el operador **sizeof** para determinar los tamaños en bytes de diferentes tipos de datos en el sistema de su computadora. Escriba los resultados en el archivo **"tamaniodatos.dat"**, para que más tarde pueda imprimir los resultados. El formato de los resultados en el archivo deben aparecer de la siguiente manera:

```
Tipo de dato             Tamanio

char                           1
unsigned char                  1
short int                      2
unsigned short int             2
int                            4
unsigned int                   4
long int                       4
unsigned long int              4
float                          4
double                         8
long double                   16
```

[*Nota:* Los tamaños correspondientes al tipo de dato, en su computadora pueden ser diferentes a los que listamos arriba.]

**11.17** En el ejercicio 7.19 escribió una simulación de software de una computadora que utilizaba un lenguaje máquina especial, llamado Lenguaje Máquina Simpletron (LMS). En la simulación, cada vez que quería ejecutar un programa en LMS, usted introducía el programa desde el teclado. Si cometió un error mientras escribía el programa en LMS, el simulador se reiniciaba y el código en SML se reintroducía. Sería bueno poder leer el programa en LMS desde un archivo, en lugar de escribirlo cada vez. Esto reduciría los errores y el tiempo para preparar la ejecución de programas en LMS.

a) Modifique el simulador que escribió en el ejercicio 7.19, para que lea programas en LMS desde un archivo especificado por el usuario desde el teclado.

b) Después de que se ejecuta el Simpletron, éste despliega en la pantalla el contenido de sus registros. Sería bueno capturar la salida en un archivo; modifique el simulador para que éste escriba su salida en un archivo, además de desplegarla en la pantalla.

# 12

# Estructuras de datos en C

## Objetivos

- Asignar y liberar memoria dinámicamente para objetos de datos.
- Formar estructuras de datos por medio de apuntadores, de estructuras autorreferenciadas y de recursividad.
- Crear y manipular listas ligadas, colas, pilas y árboles binarios.
- Comprender diversas aplicaciones importantes de las estructuras de datos ligadas.

*De muchas cosas a las que estoy atado, no he podido liberarme;*
*Y muchas cosas de las que me liberé, han vuelto a mí.*
Lee Wilson Dodd

*¿Podrías caminar un poco más rápido? dijo una merluza a un caracol,*
*"Hay una marsopa muy cerca de nosotros, y está pisándome los talones."*
Lewis Carroll

*Siempre hay lugar en la cima.*
Daniel Webster

*Continúen; sigan moviéndose.*
Thomas Morton

*Creo que nunca veré*
*un poema tan maravilloso como un árbol.*
Joyce Kilmer



## Plan general



## 12.1 Introducción

Hemos estudiado *estructuras de datos* de tamaño fijo, como arreglos con un solo subíndice, arreglos con dos subíndices y **`struct`**s. Este capítulo presenta las *estructuras de datos dinámicas* con tamaños que crecen y disminuyen en tiempo de ejecución. Las *listas ligadas* son colecciones de elementos de datos "alineados en una fila"; las inserciones y las eliminaciones se hacen en cualquier parte de una lista ligada. Las *pilas* son importantes para los compiladores y los sistemas operativos; las inserciones y las eliminaciones se hacen sólo en un extremo de la pila, esto es, en la *cima*. Las *colas* representan líneas de espera; las inserciones se hacen en la parte final (también conocida como los *talones*) de una cola, y las eliminaciones se hacen de la parte inicial (también conocida como la *cabeza*) de una cola. Los *árboles* facilitan la búsqueda y el ordenamiento de datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina. Cada una de estas estructuras de datos tiene muchas otras aplicaciones interesantes.

Explicaremos cada uno de los tipos principales de estructuras de datos e implementaremos programas que generen y manipulen dichas estructuras. En la siguiente parte del libro, la introducción a C++ y a la programación orientada a objetos, estudiaremos la abstracción de datos. Esta técnica nos permitirá construir estas estructuras de datos de manera extremadamente diferente, diseñada para producir software más fácil de mantener y reutilizar.

Éste es un capítulo desafiante. Los programas son extensos e incorporan la mayor parte de lo visto en los capítulos anteriores. Los programas son especialmente fuertes en cuanto a la manipulación de apuntadores; un tema que mucha gente considera de los más difíciles de C. El capítulo está lleno de programas prácticos que podrá utilizar en cursos más avanzados; éste incluye una gran colección de ejercicios que enfatizan las aplicaciones prácticas de las estructuras de datos.

Sinceramente esperamos que intente el proyecto principal que describimos en la sección titulada "Cómo construir su propio compilador". Usted ha estado utilizando un compilador para traducir sus programas en C a lenguaje máquina, por lo que ha podido ejecutar sus programas en su computadora. En este proyecto, realmente construirá su propio compilador. Éste leerá un archivo de instrucciones escritas en un lenguaje de alto nivel sencillo, pero poderoso, similar a las primeras versiones del popular lenguaje BASIC. Su compilador traducirá estas instrucciones a un archivo de instrucciones de Lenguaje MáquinaSimpletron (LMS). LMS es el lenguaje que aprendió en la sección especial del capítulo 7. ¡Su programa simulador Simpletron ejecutará el programa LMS que produzca en su compilador! Este proyecto le dará la maravillosa oportunidad de ejercitar casi todo lo que ha aprendido en este curso. La sección especial lo guía cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que necesitará para convertir cada tipo de instrucción del lenguaje de alto nivel en instrucciones de lenguaje máquina. Si disfruta los desafíos, podría intentar mejorar tanto el compilador como el simulador Simpletron sugeridos en los ejercicios.

## 12.2 Estructuras autorreferenciadas

Una *estructura autorreferenciada* contiene un miembro apuntador, el cual apunta hacia una estructura del mismo tipo. Por ejemplo, la definición

```
struct nodo {
   int dato;
   struct nodo *ptrSiguiente;
};
```

define un tipo, **struct nodo**. Una estructura del tipo **struct nodo** tiene dos miembros; el miembro entero **dato** y el miembro apuntador **ptrSiguiente**. El miembro **ptrSiguiente** apunta hacia la estructura de tipo **struct nodo**; una estructura del mismo tipo que la que estamos declarando aquí, y de ahí el término "estructura autorreferenciada". El miembro **ptrSiguiente** se conoce como *liga*, es decir, este miembro puede utilizarse para "unir" una estructura del tipo **struct nodo** con otra estructura del mismo tipo. Las estructuras autorreferenciadas pueden ligarse entre sí para formar estructuras de datos útiles, como listas, colas, pilas y árboles. La figura 12.1 ilustra dos objetos del tipo de estructuras autorreferenciadas ligadas para formar una lista. Observe que se coloca una diagonal (que representa un apuntador **NULL**) en el miembro liga de la segunda estructura autorreferenciada, para indicar que la liga no apunta hacia otra estructura. [*Nota:* La diagonal se utiliza sólo para efectos de ilustración; no corresponde al carácter de diagonal invertida de C.] Un apuntador **NULL** generalmente indica el final de una estructura de datos, tal como el carácter nulo indica el final de una cadena.

**Error común de programación 12.1**

*No establecer en* ***NULL*** *la liga del último nodo de una lista, puede provocar errores de ejecución.*

## 12.3 Asignación dinámica de memoria

Crear y mantener estructuras de datos dinámicas requiere de la *asignación dinámica de memoria*; es decir, la habilidad de un programa para obtener más espacio de memoria en tiempo de ejecución, para almacenar nuevos nodos, y para liberar espacio que ya no es necesario. El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria físicamente disponible en la computadora, o la virtualmente disponible en un sistema de memoria virtual. Con frecuencia, los límites son mucho más pequeños debido a que la memoria debe compartirse entre muchas aplicaciones.

Las funciones **malloc** y **free**, y el operador **sizeof** son básicos para la asignación dinámica de memoria. La función **malloc** toma como un argumento al número de bytes que van a asignarse, y devuelve un apuntador de tipo **void*** (*apuntador a* ***void***) hacia la memoria asignada. Un apuntador **void*** puede asignarse a una variable de cualquier tipo de apuntador. La función **malloc** normalmente se utiliza con el apuntador **sizeof**. Por ejemplo, la instrucción

```
ptrNuevo = malloc( sizeof( struct nodo ) );
```

evalúa a **sizeof( struct nodo )** para determinar el tamaño en bytes de una estructura del tipo **struct nodo**, para asignar una nueva área en memoria que coincida con ese número de bytes, y para almacenar un apuntador a la memoria asignada a la variable **prtNuevo**. La memoria asignada no se inicializa. Si no hay memoria disponible, **malloc** devuelve **NULL**.

La función **free** libera memoria, es decir, la memoria se devuelve al sistema para que ésta pueda reasignarse en el futuro. Para liberar memoria dinámicamente asignada por la llamada anterior a **malloc**, utilice la instrucción

```
free( ptrNuevo );
```

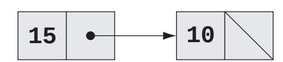



**Figura 12.1** Estructuras autorreferenciadas ligadas.

Las siguientes secciones explican las listas, pilas, colas y árboles, cada una de las cuales se crea y se mantiene por medio de estructuras autorreferenciadas y asignación dinámica de memoria.



**Tip de portabilidad 12.1**

*El tamaño de una estructura no necesariamente es la suma de los tamaños de sus miembros. Esto es así debido a los diversos requerimientos de los límites de alineación que dependen de cada máquina (vea el capítulo 10).*



**Error común de programación 12.2**

*Suponer que el tamaño de una estructura es la suma del tamaño de sus miembros, es un error lógico.*



**Buena práctica de programación 12.1**

*Utilice el operador `sizeof` para determinar el tamaño de una estructura.*



**Tip para prevenir errores 12.1**

*Cuando utilice `malloc`, evalúe la devolución de un valor de apuntador `NULL`. Imprima un mensaje de error si la memoria requerida no es asignada.*



**Error común de programación 12.3**

*No devolver la memoria asignada dinámicamente cuando ya no es necesaria, puede ocasionar que el sistema se quede sin memoria de manera prematura. A esto se le conoce en ocasiones como "fuga de memoria".*



**Buena práctica de programación 12.2**

*Cuando la memoria que se asignó dinámicamente ya no es necesaria, utilice `free` para devolverla inmediatamente al sistema.*



**Error común de programación 12.4**

*Liberar memoria no asignada dinámicamente con `malloc`, es un error.*



**Error común de programación 12.5**

*Hacer referencia a memoria que ha sido liberada, es un error.*

## 12.4 Listas ligadas

Una *lista ligada* es una colección lineal de estructuras autorreferenciadas, llamadas *nodos*, conectadas por medio de *ligas* apuntador; de aquí el término lista "ligada". Se accede a una lista ligada a través de un apuntador al primer nodo de la lista. Se accede a los nodos subsiguientes a través del miembro liga almacenado en cada nodo. Por convención, el apuntador liga del último nodo de una lista se establece en `NULL`, para marcar el final de la lista. Los datos se almacenan en una lista ligada dinámicamente; conforme es necesario, se crea cada nodo. Un nodo puede contener datos de cualquier tipo, incluso otros objetos `struct`. Las pilas y las colas también son estructuras de datos lineales y, como veremos, son versiones restringidas de listas ligadas. Los árboles son estructuras de datos no lineales.

Las listas de datos pueden almacenarse en arreglos, pero las listas ligadas proporcionan muchas ventajas. Una lista ligada es adecuada, cuando el número de elementos a representarse en la estructura de datos es impredecible. Las listas ligadas son dinámicas, por lo que la longitud de una lista puede aumentar o disminuir conforme sea necesario. Sin embargo, el tamaño de un arreglo no puede alterarse una vez que se asignó la memoria. Los arreglos pueden llenarse. Las listas ligadas sólo se llenan cuando el sistema tiene insuficiente memoria para satisfacer los requerimientos de asignación dinámica de almacenamiento.



**Tip de rendimiento 12.1**

*Un arreglo puede declararse para que contenga más elementos que los esperados; sin embargo, esto puede desperdiciar memoria. Las listas ligadas proporcionan una mejor utilización de memoria, en estas situaciones.*

Las listas ligadas pueden mantenerse ordenadas, si se inserta cada nuevo elemento en el punto adecuado de la lista.

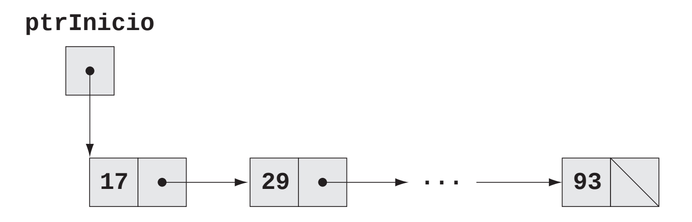
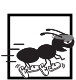



**Figura 12.2** Representación gráfica de una lista ligada.

### Tip de rendimiento 12.2

*Las inserciones y las eliminaciones en un arreglo ordenado pueden llevar demasiado tiempo; todos los elementos que siguen al elemento insertado o eliminado deben desplazarse adecuadamente.*



### Tip de rendimiento 12.3

*Los elementos de un arreglo se almacenan en memoria de manera contigua. Esto permite el acceso inmediato a un elemento de un arreglo, ya que la dirección de cualquier elemento puede calcularse directamente de acuerdo con su posición relativa al principio del arreglo. Las listas ligadas no permiten el acceso inmediato a sus elementos.*

Los nodos de una lista ligada por lo general no se almacenan contiguamente en memoria. Sin embargo, de manera lógica, los nodos de una lista ligada aparentan estar contiguos. La figura 12.2 muestra una lista ligada con diversos nodos.



### Tip de rendimiento 12.4

*Utilizar la asignación dinámica de memoria (en lugar de arreglos) para estructuras de datos que aumentan y disminuyen en tiempo de ejecución, puede ahorrar memoria. Sin embargo, recuerde que los apuntadores ocupan más espacio, y que la asignación dinámica de memoria incurre en la sobrecarga de llamadas a funciones.*

La figura 12.3 (cuya salida aparece en la figura 12.4) manipula una lista de caracteres. El programa proporciona dos opciones: 1) insertar un carácter en la lista en orden alfabético (función **`insertar`**), y 2) eliminar un carácter de la lista (función **`eliminar`**). Éste es un largo y complejo programa. A continuación daremos una explicación detallada del programa. El ejercicio 12.20 pide que se implemente una función recursiva que imprima una lista en orden inverso. El ejercicio 12.21 pide que se implemente una función recursiva que busque un elemento en particular de una lista ligada.

```
/* Figura 12.3: fig12_03.c
   Operación y mantenimiento de una lista */
#include <stdio.h>
#include <stdlib.h>

/* estructura auto_referenciada */
struct nodoLista {
   char dato; /* cada nodoLista contiene un caracter */
   struct nodoLista *ptrSiguiente;    /* apuntador al siguiente nodo */
}; /* fin de la estructura nodoLista */

typedef struct nodoLista NodoLista; /* sinónimo para la estructura nodoLista */
typedef NodoLista *ptrNodoLista;   /* sinónimo para de NodoLista* */

/* prototipos */
void insertar( ptrNodoLista *ptrS, char valor );
char eliminar( ptrNodoLista *ptrS, char valor );
int estaVacia( ptrNodoLista ptrS );
void imprimeLista( ptrNodoLista ptrActual );
```

**Figura 12.3** Inserción y eliminación de nodos en una lista. (Parte 1 de 5.)

```
void instrucciones( void );

int main()
{
   ptrNodoLista ptrInicial = NULL; /* inicialmente no existen nodos */
   int eleccion;  /* elección del usuario */
   char elemento;  /* carácter introducido por el usuario */

   instrucciones(); /* despliega el menú */
   printf( “? “ );
   scanf( “%d”, &eleccion );

   /* repite mientras el usuario no elija 3 */
   while ( eleccion != 3 ) {

      switch ( eleccion ) {

         case 1:
            printf( “Introduzca un caracter: “ );
            scanf( “\n%c”, &elemento );
            insertar( &ptrInicial, elemento ); /* inserta el elemento en la lista */
            imprimeLista( ptrInicial );
            break;

         case 2:

            /* si la lista no está vacía */
            if ( !estaVacia( ptrInicial ) ) {
               printf( “Introduzca un caracter para eliminar: “ );
               scanf( “\n%c”, &elemento );

               /* si encuentra el carácter, lo remueve */
               if ( eliminar( &ptrInicial, elemento ) ) { /* elimina elemento */
                  printf( “caracter %c eliminado.\n”, elemento );
                  imprimeLista( ptrInicial );
               } /* fin de if */
               else {
                  printf( “no se encuentra el caracter %c.\n\n”, elemento );
               } /* fin de else */

            } /* fin de if */
            else {
               printf( “La lista esta vacia.\n\n” );
            } /* fin de else */

            break;

         default:
            printf( “Opcion invalida.\n\n” );
            instrucciones();
            break;

      } /* fin de switch */
```

**Figura 12.3** Inserción y eliminación de nodos en una lista. (Parte 2 de 5.)

```
      printf( “? “ );
      scanf( “%d”, &eleccion );
   } /* fin de while */

   printf( “Fin de la ejecucion.\n” );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* despliega las instrucciones del programa para el usuario */
void instrucciones( void )
{
   printf( “Introduzca su eleccion:\n”
      “   1 para insertar un elemento en la lista.\n”
      “   2 para eliminar un elemento de la lista.\n”
      “   3 para terminar.\n” );
} /* fin de la función instrucciones */

/* Inserta un nuevo valor dentro de la lista en orden  */
void insertar( ptrNodoLista *ptrS, char valor )
{
   ptrNodoLista ptrNuevo;    /* apuntador a un nuevo nodo */
   ptrNodoLista ptrAnterior; /* apuntador a un nodo previo de la lista */
   ptrNodoLista ptrActual;   /* apuntador al nodo actual de la lista */

   ptrNuevo = malloc( sizeof( NodoLista ) ); /* crea un nodo */

   if ( ptrNuevo != NULL ) {    /* es espacio disponible */
      ptrNuevo->dato = valor;   /* coloca el valor en el nodo */
      ptrNuevo->ptrSiguiente = NULL; /* el nodo no se liga a otro nodo */

      ptrAnterior = NULL;
      ptrActual = *ptrS;

      /* ciclo para localizar la ubicación correcta en la lista */
      while ( ptrActual != NULL && valor > ptrActual->dato ) {
         ptrAnterior = ptrActual;              /* entra al ...   */
         ptrActual = ptrActual->ptrSiguiente;  /* ... siguiente nodo */
      } /* fin de while */

      /* inserta un nuevo nodo al principio de la lista */
      if ( ptrAnterior == NULL ) {
         ptrNuevo->ptrSiguiente = *ptrS;
         *ptrS = ptrNuevo;
      } /* fin de if */
      else { /* inserta un nuevo nodo entre ptrAnterior y ptrActual */
         ptrAnterior->ptrSiguiente = ptrNuevo;
         ptrNuevo->ptrSiguiente = ptrActual;
      } /* fin de else */

   } /* fin de if */
   else {
      printf( “No se inserto %c. No hay memoria disponible.\n”, valor );
```

**Figura 12.3** Inserción y eliminación de nodos en una lista. (Parte 3 de 5.)

```
   } /* fin de else */

} /* fin de la función insertar */

/* Elimina un elemento de la lista */
char eliminar( ptrNodoLista *ptrS, char valor )
{
   ptrNodoLista ptrAnterior; /* apuntador a un nodo previo de la lista */
   ptrNodoLista ptrActual;   /* apuntador al nodo actual de la lista */
   ptrNodoLista tempPtr;     /* apuntador a un nodo temporal */

   /* elimina el primer nodo */
   if ( valor == ( *ptrS )->dato ) {
      tempPtr = *ptrS; /* almacena el nodo a eliminar */
      *ptrS = ( *ptrS )->ptrSiguiente; /* desata el nodo */
      free( tempPtr ); /* libera el nodo desatado */
      return valor;
   } /* fin de if */
   else {
      ptrAnterior = *ptrS;
      ptrActual = ( *ptrS )->ptrSiguiente;

      /* ciclo para localizar la ubicación correcta en la lista */
      while ( ptrActual != NULL && ptrActual->dato != valor ) {
         ptrAnterior = ptrActual;             /* entra al ...    */
         ptrActual = ptrActual->ptrSiguiente; /* ... siguiente nodo */
      } /* fin de while */

      /* elimina el nodo de ptrActual */
      if ( ptrActual != NULL ) {
         tempPtr = ptrActual;
         ptrAnterior->ptrSiguiente = ptrActual->ptrSiguiente;
         free( tempPtr );
         return valor;
      } /* fin de if */

   } /* fin de else */

   return '\0';

} /* fin de la función eliminar */

/* Devuelve 1 si la lista está vacía, de lo contrario, 0  */
int estaVacia( ptrNodoLista ptrS )
{
   return ptrS == NULL;

} /* fin de la función function estaVacia */

/* Imprime la lista */
void imprimeLista( ptrNodoLista ptrActual )
{

   /* si la lista está vacía */
   if ( ptrActual == NULL ) {
```

**Figura 12.3** Inserción y eliminación de nodos en una lista. (Parte 4 de 5.)

```
         printf( “La lista esta vacia.\n\n” );
      } /* fin de if */
      else {
         printf( “La lista es:\n” );

         /* mientras no sea el final de la lista */
         while ( ptrActual != NULL ) {
            printf( “%c –> “, ptrActual->dato );
            ptrActual = ptrActual->ptrSiguiente;
         } /* fin de while */

         printf( “NULL\n\n” );
      } /* fin de else */

} /* fin de la función imprimeLista */
```

**Figura 12.3** Inserción y eliminación de nodos en una lista. (Parte 5 de 5.)

```
Introduzca su eleccion:
   1 para insertar un elemento en la lista.
   2 para eliminar un elemento de la lista.
   3 para terminar.
? 1
Introduzca un caracter: B
La lista es:
B --> NULL

? 1
Introduzca un caracter: A
La lista es:
A --> B --> NULL

? 1
Introduzca un caracter: C
La lista es:
A --> B --> C --> NULL

? 2
Introduzca un caracter para eliminar: D
no se encuentra el caracter D.

? 2
Introduzca un caracter para eliminar: B
caracter B eliminado.
La lista es:
A --> C --> NULL

? 2
Introduzca un caracter para eliminar: C
caracter C eliminado.
La lista es:
A --> NULL
```

**Figura 12.4** Salida de ejemplo del programa de la figura 12.3. (Parte 1 de 2.)

```
? 2
Introduzca un caracter para eliminar: A
caracter A eliminado.
La lista esta vacia.

? 4
Opcion invalida.

Introduzca su eleccion:
   1 para insertar un elemento en la lista.
   2 para eliminar un elemento de la lista.
   3 para terminar.
? 3
Fin de la ejecucion.
```

**Figura 12.4** Salida de ejemplo del programa de la figura 12.3. (Parte 2 de 2.)

Las funciones primarias de las listas ligadas son **`insertar`** (líneas 94 a 130) y **`eliminar`** (líneas 133 a 168). A la función **`estaVacia`** (líneas 171 a 175) se le llama *función predicado*; ésta no altera la lista de manera alguna, lo que hace es determinar si la lista está vacía (es decir, si el apuntador al primer nodo es **`NULL`**). Si la lista está vacía, se devuelve 1; de lo contrario, se devuelve 0. La función **`imprimeLista`** (líneas 178 a 197) imprime la lista.

Los caracteres se insertan en la lista en orden alfabético. La función **`insertar`** (líneas 94 a 130) recibe la *dirección* de la lista y un carácter a insertar. La dirección de la lista es necesaria cuando va a insertarse un valor al principio de la lista. Proporcionar la dirección de la lista le permite a la lista (es decir, al apuntador al primer nodo de la lista) que se le modifique a través de una llamada por referencia. Debido a que la lista misma es un apuntador (a su primer elemento), pasar la dirección de la lista crea un *apuntador a un apuntador* (es decir, una *doble indirección*). Ésta es una noción compleja y requiere una programación cuidadosa. Los pasos para insertar un carácter en la lista son los siguientes (vea la figura 12.5):

1. Cree un nodo mediante una llamada a **`malloc`**, que asigne a **`ptrNuevo`** la dirección de la memoria asignada (línea 100), que asigne el carácter a insertar en **`ptrNuevo->dato`** (línea 103), y que asigne **`NULL`** a **`ptrNuevo->ptrSiguiente`** (línea 104).
2. Inicialice **`ptrAnterior`** en **`NULL`** (línea 106) y a **`ptrActual`** en **`*ptrS`** (línea 107), es decir, al apuntador al inicio de la lista. El apuntador **`ptrAnterior`** almacena la ubicación del nodo anterior al punto de inserción, y el apuntador **`ptrActual`** almacena la ubicación del nodo siguiente al punto de inserción.
3. Mientras **`ptrActual`** no sea **`NULL`** y el valor a insertar sea mayor que **`ptrActual->dato`** (línea 110), asigne **`ptrActual`** a **`ptrAnterior`** (línea 111) y adelante **`ptrActual`** al siguiente nodo de la lista (línea 112). Esto ubica el punto de inserción para el valor.
4. Si **`ptrAnterior`** es **`NULL`** (línea 116), inserte el nuevo nodo como el primero de la lista (líneas 117 y 118). Asigne **`*prtS`** a **`ptrNuevo->ptrSiguiente`** (el nuevo nodo liga los puntos al primer nodo anterior) y asigne **`ptrNuevo`** a **`*ptrS`** (**`*ptrS`** apunta al nuevo nodo). De lo contrario, si **`ptrAnterior`** no es **`NULL`**, el nuevo nodo se inserta en su lugar (líneas 121 y 122). Asigne **`ptrNuevo`** al **`ptr->ptrSiguiente`** anterior (el nodo anterior apunta al nuevo nodo) y asigne **`ptrActual`** a **`ptrNuevo->ptrSiguiente`** (el nuevo nodo liga los puntos al nodo actual).



**Tip para prevenir errores 12.2**

*Asigne **`NULL`** al miembro liga de un nuevo nodo. Los apuntadores deben inicializarse antes de que se utilicen.*

La figura 12.5 ilustra la inserción de un nodo que contiene el carácter **`'C'`** en una lista ordenada. La parte a) de la figura muestra la lista y el nuevo nodo antes de la inserción del nuevo nodo. La parte b) muestra el resultado de la inserción del nuevo nodo. Los apuntadores reasignados son las flechas punteadas.

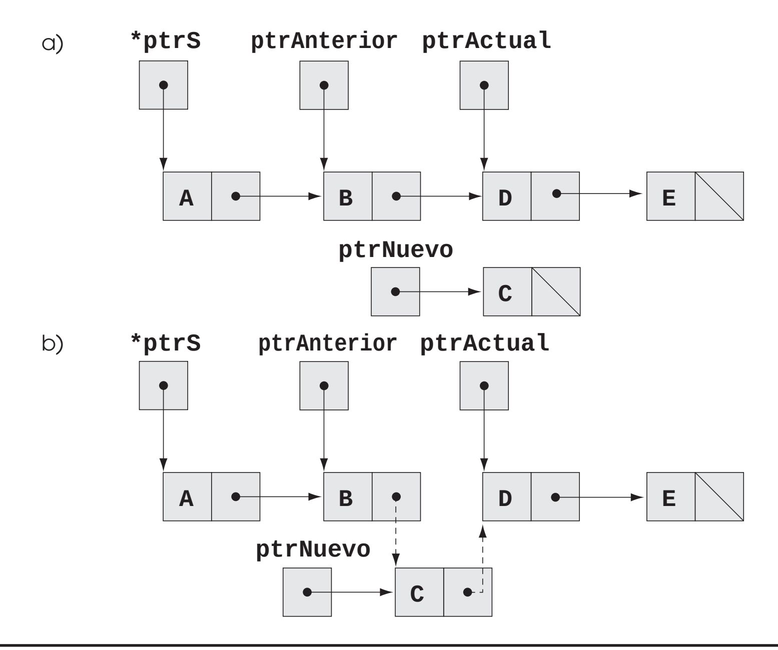



**Figura 12.5** Inserción ordenada de un nodo en una lista.

La función **eliminar** (líneas 137 a 168) recibe la dirección del apuntador al inicio de la lista y un carácter que debe eliminarse. Los pasos para eliminar un carácter de la lista son los siguientes:

1. Si el carácter a eliminar coincide con el carácter del primer nodo de la lista (línea 140), asigne ***ptrS** a **ptrTemp** (**ptrTemp** se utilizará para liberar la memoria innecesaria), asigne **(*ptrS)->ptrSiguiente** a ***ptrS** (ahora ***ptrS** apunta al segundo nodo de la lista), libere la memoria apuntada por **ptrTemp**, y devuelva el carácter que se eliminó.

2. De lo contrario, inicialice **ptrAnterior** en ***ptrS**, e inicialice **ptrActual** en **(*ptrS)->ptrSiguiente** (líneas 147 y 148).

3. Mientras **ptrActual** no sea **NULL** y el valor a eliminar no sea igual a **ptrActual->dato** (línea 151), asigne **ptrActual** a **ptrAnterior** (línea 152), y asigne **ptrActual->ptrSiguiente** a **ptrActual** (línea 153). Esto ubica el carácter a eliminar, si éste se encuentra en la lista.

4. Si **ptrActual** no es **NULL** (línea 157), asigne **ptrActual** a **ptrTemp** (línea 158), asigne **ptrActual->ptrSiguiente** a **ptrAnterior->ptrSiguiente** (línea 159), libere el nodo apuntado por **ptrTemp** (línea 160), y devuelva el carácter eliminado de la lista (línea 161). Si **ptrActual** es **NULL**, devuelva el carácter nulo (**'\0'**) para indicar que el carácter a eliminar no se encontró en la lista (línea 166).

La figura 12.6 ilustra la eliminación de un nodo de una lista ligada. La parte a) de la figura muestra la lista ligada después de la operación de inserción anterior. La parte b) muestra la reasignación del elemento liga de **ptrAnterior** y la asignación de **ptrActual** a **ptrTemp**. El apuntador **ptrTemp** se utiliza para liberar la memoria asignada para almacenar **'C'**.

La función **imprimeLista** (líneas 178 a 197) recibe como argumento un apuntador al inicio de la lista y hace referencia al apuntador como **ptrActual**. La función primero determina si la lista está vacía (líneas 182 a 184) y, si es así, imprime **"La lista esta vacia."**, y termina. De lo contrario, imprime el dato de la lista (líneas 185 a 195). Mientras **ptrActual** no sea **NULL**, **ptrActual->dato** es impreso por la función, y **ptrActual->ptrSiguiente** se asigna a **ptrActual**. Observe que si la liga del último nodo de la lista no es **NULL**, el algoritmo de impresión intentará imprimir más allá del final de la lista, y se generará un error. El algoritmo de impresión es idéntico para listas ligadas, pilas y colas.

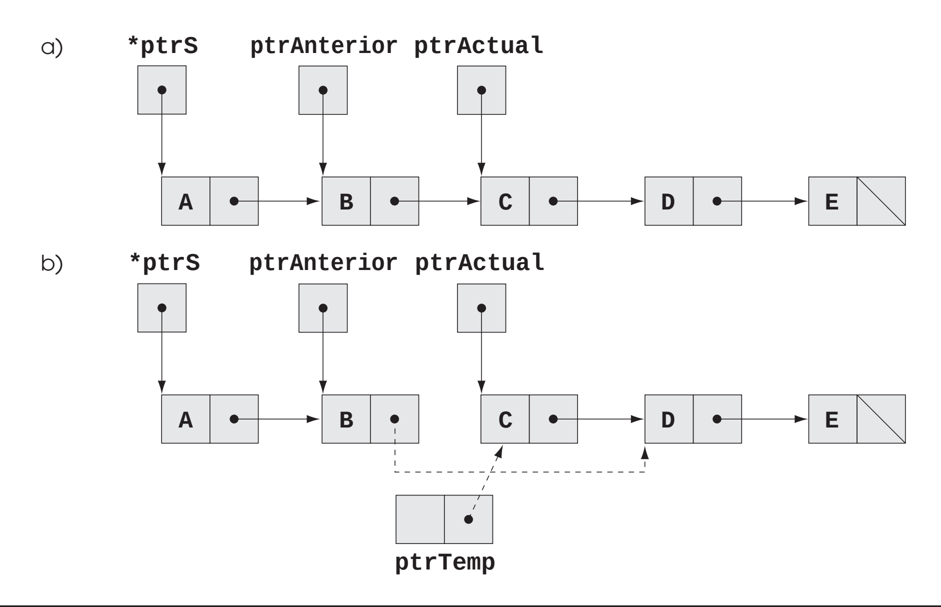



**Figura 12.6** Eliminación de un nodo de una lista.

## 12.5 Pilas

Una *pila* es una versión restringida de una lista ligada. Los nuevos nodos pueden añadirse y eliminarse de una pila sólo en la cima. Por esta razón, a una pila se le conoce como una estructura de datos *última en entrar*, *primera en salir* (*UEPS*). Se hace referencia a una pila por medio de un apuntador hacia el elemento en la cima de la pila. El miembro liga del último nodo de la pila se establece en NULL para indicar el fondo de la pila.

La figura 12.7 muestra una pila con diversos nodos. Observe que las pilas y las listas ligadas se representan de manera idéntica. La diferencia entre las pilas y las listas ligadas es que las inserciones y las eliminaciones pueden ocurrir en cualquier parte de la lista ligada, mientras que en una pila, dichas operaciones se realizan sólo en la cima de ésta.



### Error común de programación 12.6

*No establecer en* **`NULL`** *la liga del nodo del fondo de una pila puede ocasionar errores de ejecución.*

Las funciones básicas que se utilizan para manipular una pila son empujar y sacar. La función empujar crea un nuevo nodo y lo coloca en la cima de la pila. La función sacar elimina un nodo de la cima de la pila, libera la memoria que estaba asignada al nodo eliminado y devuelve el valor eliminado.

La figura 12.8, cuya salida aparece en la figura 12.9, implementa una pila simple de enteros. El programa proporciona tres opciones: 1) introducir un valor en la pila (función **`empujar`**), 2) eliminar un valor de la pila (función **`sacar`**), y 3) finalizar el programa.

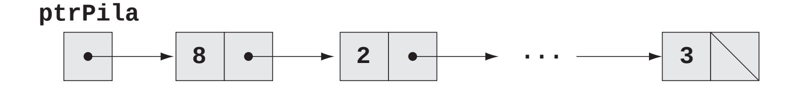



**Figura 12.7** Representación gráfica de una pila.

```
/* Figura 12.8: fig12_08.c
   programa de pila dinámica */
#include <stdio.h>
```

**Figura 12.8** Un programa sobre una pila simple. (Parte 1 de 4.)

```
#include <stdlib.h>

/* estructura auto-referenciada */
struct nodoPila {
   int dato;                      /* define un dato como int */
   struct nodoPila *ptrSiguiente; /* apuntador a nodoPila */
}; /* fin de la estructura nodoPila */

typedef struct nodoPila NodoPila; /* sinónimo de la estructura nodoPila */
typedef NodoPila *ptrNodoPila;    /* sinónimo para NodoPila* */

/* prototipos */
void empujar( ptrNodoPila *ptrCima, int info );
int sacar( ptrNodoPila *ptrCima );
int estaVacia( ptrNodoPila ptrCima );
void imprimePila( ptrNodoPila ptrActual );
void instrucciones( void );

/* la función main comienza la ejecución del programa */
int main()
{
   ptrNodoPila ptrPila = NULL; /* apunta al tope de la pila */
   int eleccion; /* elección de menú del usuario */
   int valor;    /* entrada int del usuario */

   instrucciones(); /* despliega el menú */
   printf( “? “ );
   scanf( “%d”, &eleccion );

   /* mientras el usuario no introduzca 3 */
   while ( eleccion != 3 ) {

      switch ( eleccion ) {

         /* empuja el valor dentro de la pila */
         case 1:
            printf( “Introduzca un entero: “ );
            scanf( “%d”, &valor );
            empujar( &ptrPila, valor );
            imprimePila( ptrPila );
            break;

         /* saca el valor de la pila */
         case 2:

            /* si la pila no está vacía */
            if ( !estaVacia( ptrPila ) ) {
               printf( “El valor sacado es %d.\n”, sacar( &ptrPila ) );
            } /* fin de if */

            imprimePila( ptrPila );
            break;

         default:
            printf( “Eleccion no valida.\n\n” );
```

**Figura 12.8** Un programa sobre una pila simple. (Parte 2 de 4.)

```
            instrucciones();
            break;

      } /* fin de switch */

      printf( "? " );
      scanf( "%d", &eleccion );
   } /* fin de while */

   printf( "Fin del programa.\n" );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* despliega las instrucciones del programa para el usuario */
void instrucciones( void )
{
   printf( "Introduzca su eleccion:\n"
      "1 para empujar un valor dentro de la pila\n"
      "2 para sacar un valor de la pila\n"
      "3 para terminar el programa\n" );
} /* fin de la función instrucciones */

/* Inserta un nodo en la cima de la pila */
void empujar( ptrNodoPila *ptrCima, int info )
{
   ptrNodoPila ptrNuevo; /* apuntador al nuevo nodo */

   ptrNuevo = malloc( sizeof( NodoPila ) );

   /* inserta el nodo en la cima de la pila */
   if ( ptrNuevo != NULL ) {
      ptrNuevo->dato = info;
      ptrNuevo->ptrSiguiente = *ptrCima;
      *ptrCima = ptrNuevo;
   } /* fin de if */
   else { /* no queda espacio disponible */
      printf( "%d no se inserto. Memoria insuficiente.\n", info );
   } /* fin de else */

} /* fin de la función empujar */

/* Elimina un nodo de la cima de la pila */
int sacar( ptrNodoPila *ptrCima )
{
   ptrNodoPila ptrTemp; /* apuntador a un nodo temporal */
   int valorElim; /* valor del nodo */

   ptrTemp = *ptrCima;
   valorElim = ( *ptrCima )->dato;
   *ptrCima = ( *ptrCima )->ptrSiguiente;
   free( ptrTemp );

   return valorElim;
```

**Figura 12.8** Un programa sobre una pila simple. (Parte 3 de 4.)

```

} /* fin de la función sacar */

/* Imprime la pila */
void imprimePila( ptrNodoPila ptrActual )
{

   /* si la pila está vacía */
   if ( ptrActual == NULL ) {
      printf( “La pila está vacia.\n\n” );
   } /* fin de if */
   else {
      printf( “La pila es:\n” );

      /* mientras no sea el final de la pila */
      while ( ptrActual != NULL ) {
         printf( “%d —> “, ptrActual->dato );
         ptrActual = ptrActual->ptrSiguiente;
      } /* fin de while */

      printf( “NULL\n\n” );
   } /* fin de else */

} /* fin de la función imprimePila */

/* Devuelve 1 si la pila está vacía, de lo contrario 0 */
int estaVacia( ptrNodoPila ptrCima )
{
   return ptrCima == NULL;

} /* fin de la función estaVacia */
```

**Figura 12.8** Un programa sobre una pila simple. (Parte 4 de 4.)

```
Introduzca su eleccion:
1 para empujar un valor dentro de la pila
2 para sacar un valor de la pila
3 para terminar el programa
? 1
Introduzca un entero: 5
La pila es:
5 --> NULL

? 1
Introduzca un entero: 6
La pila es:
6 --> 5 --> NULL

? 1
Introduzca un entero: 4
La pila es:
4 --> 6 --> 5 --> NULL
```

**Figura 12.9** Salida de ejemplo del programa correspondiente a la figura 12.8. (Parte 1 de 2.)

```
? 2
El valor sacado es 4.
La pila es:
6 --> 5 --> NULL

? 2
El valor sacado es 6.
La pila es:
5 --> NULL

? 2
El valor sacado es 5.
La pila esta vacia.

? 4
Eleccion no valida.

Introduzca su eleccion:
1 para empujar un valor dentro de la pila
2 para sacar un valor de la pila
3 para terminar el programa
? 3
Fin del programa.
```

**Figura 12.9** Salida de ejemplo del programa correspondiente a la figura 12.8. (Parte 2 de 2.)

La función **empujar** (líneas 84 a 100) coloca un nuevo nodo en la cima de la pila. La función consiste en tres pasos:

1. Crea un nuevo nodo, llamando a **malloc** y asigna a **ptrNuevo** la ubicación de la memoria asignada (línea 88).
2. Asigna a **ptrNuevo->dato** el valor a colocarse en la pila (línea 92), y asigna ***ptrCima** (el apuntador cima de la pila) a **ptrNuevo->ptrSiguiente** (línea 93); el miembro liga de **ptrNuevo** ahora apunta al nodo cima anterior.
3. Asigna **ptrNuevo** a ***ptrCima** (línea 94); ***ptrCima** ahora apunta a la nueva cima de la pila.

Las manipulaciones que involucran a ***ptrCima** modifican el valor de **ptrPila** en **main**. La figura 12.10 muestra la función empujar. La parte a) de la figura muestra la pila y el nuevo nodo antes de la opera-

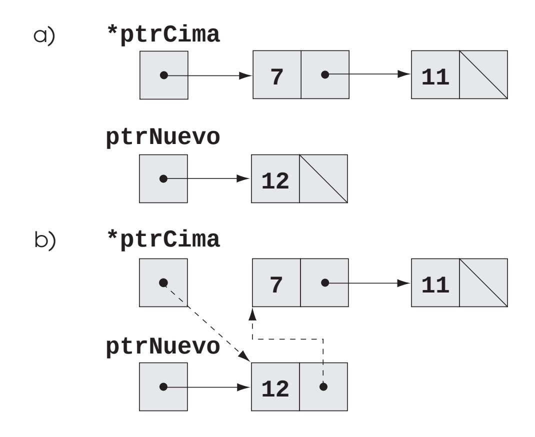



**Figura 12.10** Operación **empujar**.

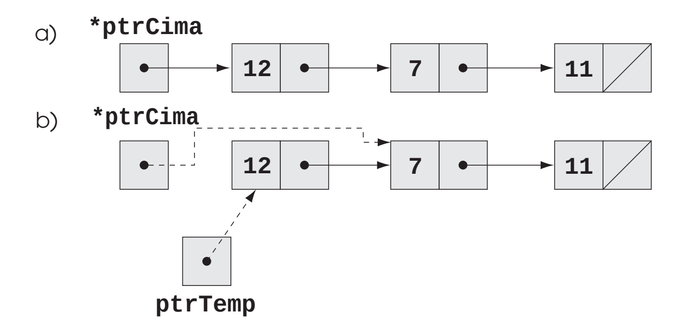



**Figura 12.11** Operación **pop** (sacar).

ción empujar. Las flechas punteadas de la parte b) muestran los pasos 2 y 3 de la operación empujar que permite que el nodo que contiene 12 se convierta en la nueva cima de la pila.

La función sacar (líneas 103 a 115) elimina un nodo de la cima de la pila. Observe que **main** determina si la pila está vacía, antes de llamar a sacar. La operación sacar consiste en cinco pasos:

1. Asigna ***ptrCima** a **ptrTemp** (línea 108); **ptrTep** se utilizará para liberar memoria innecesaria.
2. Asigna **(*ptrCima)->dato** a **valorElim** (línea 109) para guardar el valor del nodo cima.
3. Asigna **(*ptrCima)->ptrSiguiente** a ***ptrCima** (línea 110), por lo que ***ptrCima** contiene la dirección del nuevo nodo cima.
4. Libera la memoria apuntada por **ptrTemp** (línea 111).
5. Devuelve **valorElim** a la función que hizo la llamada (línea 113).

La figura 12.11 muestra la función sacar. La parte a) muestra la pila, antes de la operación empujar anterior. La parte b) muestra a **ptrTemp** apuntando al primer nodo de la pila y a **ptrCima** apuntando al segundo nodo de la pila. La función **free** se utiliza para liberar la memoria apuntada por **ptrTemp**.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, siempre que se hace una llamada a una función, la función llamada debe saber cómo regresar a quien la llamó, por lo que la dirección de retorno se introduce en una pila. Si se suscita una serie de llamadas a una función, los valores de retorno sucesivos se colocan en la pila en el orden de último en entrar, primero en salir, por lo que cada función puede volver a quien la llamó. Las pilas soportan llamadas recursivas a funciones, de la misma manera que soportan llamadas convencionales no recursivas.

Las pilas contienen el espacio creado para variables automáticas en cada invocación a una función. Cuando la función regresa a quien la llamó, el espacio de las variables automáticas de esa función se elimina de la pila, y esas variables ya no son conocidas por el programa. Los compiladores utilizan las pilas en el proceso de evaluación de expresiones y de generación de código en lenguaje máquina. Los ejercicios analizan diversas aplicaciones de las pilas.

## 12.6 Colas

Otra estructura de datos común es la *cola*. Una cola es parecida a una fila para pagar en un supermercado; a la primera persona de la fila se le atiende primero, y los demás clientes entran a la fila sólo al final de ella, y esperan a que se les atienda. Los nodos de una cola se eliminan sólo de la *cabeza de la cola*, y se insertan sólo en los *talones* de ella. Por esta razón, a una cola se le conoce como una estructura de datos *primera en entrar, primera en salir* (*PEPS*). Las operaciones de insertar y eliminar se conocen como agregar en la cola y retirar de la cola.

Las colas tienen muchas aplicaciones en sistemas de cómputo. Muchas computadoras sólo tienen un procesador, por lo que sólo es posible atender a un usuario a la vez. Las entradas de los demás usuarios se colocan en una cola. Cada entrada avanza gradualmente desde el frente de la cola, conforme los usuarios reciben servicio. La entrada del frente de la cola es la siguiente en recibir servicio.

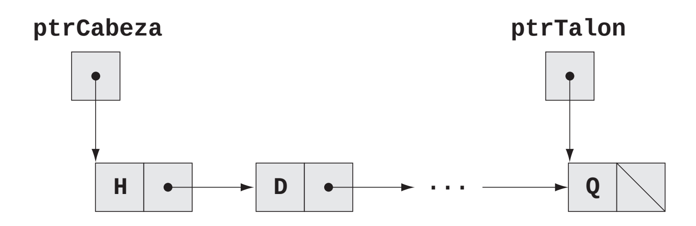



**Figura 12.12** Representación gráfica de una cola.

Las colas también se utilizan para apoyar las colas de impresión. Un ambiente multiusuario puede tener una única impresora, y muchos usuarios podrían estar generando resultados para impresión. Si la impresora está ocupada, es posible que otras salidas se estén generando, las cuales se envían a disco, donde esperan en una cola hasta que la impresora esté disponible.

Los paquetes de información también esperan en colas correspondientes a redes de computadoras. Cada vez que llega un paquete a un nodo de la red, éste debe rutearse al siguiente nodo de la red, a través de la ruta hacia el destino final del paquete. El nodo ruteador envía un paquete a la vez, por lo que los demás paquetes se colocan en la cola hasta que el ruteador los llame. La figura 12.12 muestra una cola con diversos nodos. Observe los apuntadores hacia la cabeza de la cola y hacia los talones de ésta.

### Error común de programación 12.7

*No establecer en **NULL** la liga del último nodo de una cola, puede ocasionar errores de ejecución.*

La figura 12.13, cuya salida aparece en la figura 12.14, realiza manipulaciones a una cola. El programa proporciona diversas opciones: insertar un nodo en la cola (función **agregar**, **enque**), eliminar un nodo de la cola (función **retirar**, **dequeue**), y finalizar el programa.

```
/* Figura 12.13: fig12_13.c
   Operación y mantenimiento de una cola */

#include <stdio.h>
#include <stdlib.h>

/* estructura autorreferenciada */
struct nodoCola {
   char dato;                      /* define dato como un char */
   struct nodoCola *ptrSiguiente; /* apuntador nodoCola */
}; /* fin de la estructura nodoCola */

typedef struct nodoCola NodoCola;
typedef NodoCola *ptrNodoCola;

/* prototipos de las funciones */
void imprimeCola( ptrNodoCola ptrActual );
int estaVacia( ptrNodoCola ptrCabeza );
char retirar( ptrNodoCola *ptrCabeza, ptrNodoCola *ptrTalon );
void agregar( ptrNodoCola *ptrCabeza, ptrNodoCola *ptrTalon,
               char valor );
void instrucciones( void );

/* la función main comienza la ejecución del programa */
int main()
{
```

**Figura 12.13** Procesamiento de una cola. (Parte 1 de 4.)

```
   ptrNodoCola ptrCabeza = NULL; /* incializa ptrCabeza */
   ptrNodoCola ptrTalon = NULL;  /* incializa ptrTalon */
   int eleccion;                 /* elección de menú del usuario */
   char elemento;                /* entrada char del usuario */

   instrucciones(); /* despliega el menú */
   printf( “? “ );
   scanf( “%d”, &eleccion );

   /* mientras el usuario no introduzca 3 */
   while ( eleccion != 3 ) {

      switch( eleccion ) {

         /* agrega el valor */
         case 1:
            printf( “Introduzca un caracter: “ );
            scanf( “\n%c”, &elemento );
            agregar( &ptrCabeza, &ptrTalon, elemento );
            imprimeCola( ptrCabeza );
            break;

         /* retira el valor */
         case 2:

            /* si la cola no está vacía */
            if ( !estaVacia( ptrCabeza ) ) {
               elemento = retirar( &ptrCabeza, &ptrTalon );
               printf( “se desenfilo %c.\n”, elemento );
            } /* fin de if */

            imprimeCola( ptrCabeza );
            break;

         default:
            printf( “Eleccion no valida.\n\n” );
            instrucciones();
            break;

      } /* fin de switch */

      printf( “? “ );
      scanf( “%d”, &eleccion );
   } /* fin de while */

   printf( “Fin de programa.\n” );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* despliega las instrucciones del programa para el usuario */
void instrucciones( void )
{
   printf ( “Introduzca su eleccion:\n”
```

**Figura 12.13** Procesamiento de una cola. (Parte 2 de 4.)

```
            “   1 para retirar un elemento a la cola\n”
            “   2 para eliminar un elemento de la cola\n”
            “   3 para terminar\n” );
} /* fin de la función instrucciones */

/* inserta un nodo al final de la cola */
void agregar( ptrNodoCola *ptrCabeza, ptrNodoCola *ptrTalon,
               char valor )
{
   ptrNodoCola ptrNuevo; /* apuntador a un nuevo nodo */

   ptrNuevo = malloc( sizeof( NodoCola ) );

   if ( ptrNuevo != NULL ) { /* es espacio disponible */
      ptrNuevo->dato = valor;
      ptrNuevo->ptrSiguiente = NULL;

      /* si está vacía inserta un nodo en la cabeza */
      if ( estaVacia( *ptrCabeza ) ) {
         *ptrCabeza = ptrNuevo;
      } /* fin de if */
      else {
         ( *ptrTalon )->ptrSiguiente = ptrNuevo;
      } /* fin de else */

      *ptrTalon = ptrNuevo;
   } /* fin de if */
   else {
      printf( “no se inserto %c. No hay memoria disponible.\n”, valor );
   } /* fin de else */

} /* fin de la función agregar */

/* elimina el nodo de la cabeza de la cola */
char retirar( ptrNodoCola *ptrCabeza, ptrNodoCola *ptrTalon )
{
   char valor;          /* valor del nodo */
   ptrNodoCola tempPtr; /* apuntador a un nodo temporal */

   valor = ( *ptrCabeza )->dato;
   tempPtr = *ptrCabeza;
   *ptrCabeza = ( *ptrCabeza )->ptrSiguiente;

   /* si la cola está vacía */
   if ( *ptrCabeza == NULL ) {
      *ptrTalon = NULL;
   } /* fin de if */

   free( tempPtr );

   return valor;

} /* fin de la función retirar */

/* Devuelve 1 si la cola está vacía, de lo contrario devuelve 0 */
```

**Figura 12.13** Procesamiento de una cola. (Parte 3 de 4.)

```
int estaVacia( ptrNodoCola ptrCabeza )
{
   return ptrCabeza == NULL;

} /* fin de la función estaVacia */

/* Imprime la cola */
void imprimeCola( ptrNodoCola ptrActual )
{

   /* si la cola está vacía */
   if ( ptrActual == NULL ) {
      printf( “La cola esta vacia.\n\n” );
   } /* fin de if */
   else {
      printf( “La cola es:\n” );

      /* mientras no sea el final de la cola */
      while ( ptrActual != NULL ) {
         printf( “%c –> “, ptrActual->dato );
         ptrActual = ptrActual->ptrSiguiente;
      } /* fin de while */

      printf( “NULL\n\n” );
   } /* fin de else */

} /* fin de la función imprimeCola */
```

**Figura 12.13** Procesamiento de una cola. (Parte 4 de 4.)

```
Introduzca su elección:
   1 para retirar un elemento a la cola
   2 para eliminar un elemento de la cola
   3 para terminar
? 1
Introduzca un caracter: A
La cola es:
A --> NULL

? 1
Introduzca un caracter: B
La cola es:
A --> B --> NULL

? 1
Introduzca un caracter: C
La cola es:
A --> B --> C --> NULL

? 2
Se desenfilo A.
La cola es:
B --> C --> NULL
```

**Figura 12.14** Salida de ejemplo del programa correspondiente a la figura 12.13. (Parte 1 de 2.)

```
? 2
se desenfilo B.
La cola es:
C --> NULL

? 2
se desenfilo C.
La cola esta vacia.

? 2
La cola esta vacia.

? 4
Eleccion no valida.

Introduzca su eleccion:
   1 para retirar un elemento a la cola
   2 para eliminar un elemento de la cola
   3 para terminar
? 3
Fin de programa.
```

**Figura 12.14** Salida de ejemplo del programa correspondiente a la figura 12.13. (Parte 2 de 2.)

La función **`agregar`** (líneas 88 a 113) recibe tres argumentos de **`main`**: la dirección de un apuntador hacia la cabeza de la cola, la dirección del apuntador hacia los talones de la cola, y el valor a insertar en la cola. La función consiste en tres pasos:

1. Crear un nuevo nodo: llamar a **`malloc`**, asignar a **`ptrNuevo`** la ubicación de la memoria asignada (línea 93), asignar a **`ptrNuevo->dato`** el valor a insertar en la cola, y asignar **`NULL`** a **`ptrNuevo->ptrSiguiente`** (línea 97).
2. Si la cola está vacía (línea 100), asigna **`ptrNuevo`** a **`*ptrCabeza`** (línea 101); de lo contrario, asigna el apuntador **`ptrNuevo`** a **`(*ptrTalon)->ptrSiguiente`** (línea 104).
3. Asigna **`ptrNuevo`** a **`*ptrTalon`** (línea 107).

La figura 12.15 ilustra una operación **`agregar`**. La parte a) muestra la cola y el nuevo nodo, antes de la operación. Las flechas punteadas de la parte b) ilustran los pasos 2 y 3 de la función **`agregar`** que permiten que se adicione un nuevo nodo al final de una cola que no está vacía.

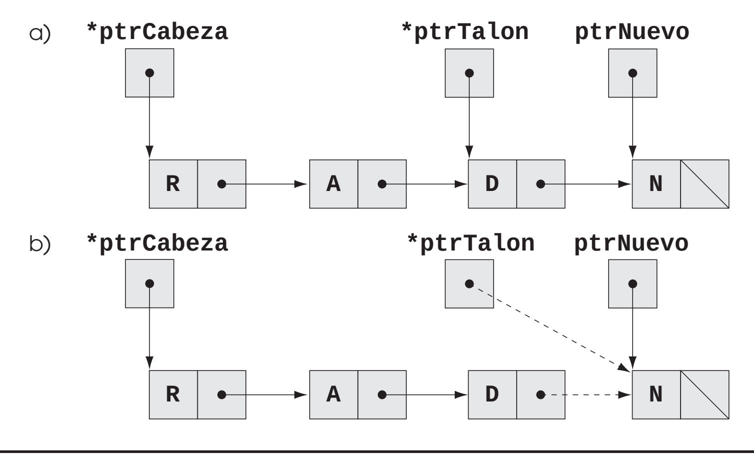



**Figura 12.15** Operación **`agregar`**.

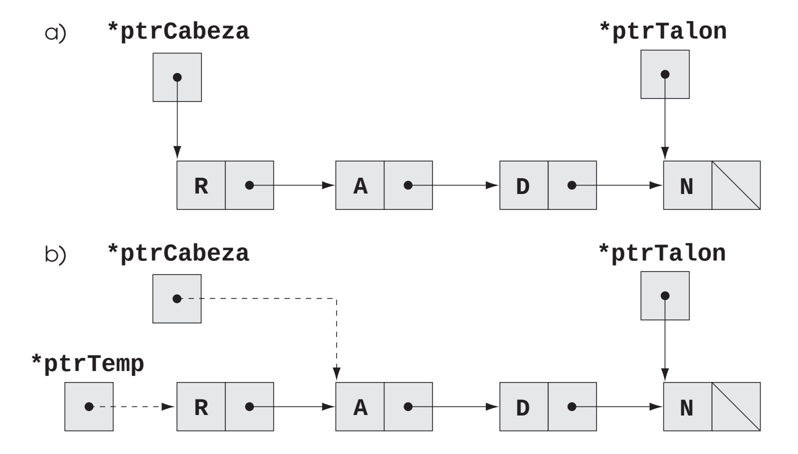



**Figura 12.16** Operación `retirar`.

La función **`retirar`** (líneas 116 a 134) recibe como argumentos la dirección del apuntador hacia la cabeza de la cola y la dirección del apuntador hacia los talones de la cola, y elimina el primer nodo de la cola. La operación **`eliminar`** consiste en seis pasos:

1. Asigna **`(*ptrCabeza)->dato`** a **`valor`**, para guardar el dato (línea 121).
2. Asigna **`*ptrCabeza`** a **`ptrTemp`** (línea 122), el cual se utilizará para liberar la memoria innecesaria.
3. Asigna **`(*ptrCabeza)->ptrSiguiente`** a **`*ptrCabeza`** (línea 123), por lo que **`*ptrCabeza`** ahora apunta hacia el nuevo primer nodo de la cola.
4. Si **`*ptrCabeza`** es **`NULL`** (línea 126), asigna **`NULL`** a **`*ptrTalon`** (línea 127).
5. Libera la memoria apuntada por **`ptrTemp`** (línea 130).
6. Devuelve **`valor`** a la función que hizo la llamada (línea 132).

La figura 12.16 ilustra la función **`retirar`**. La parte a) muestra la cola antes de la operación **`agregar`** anterior. La parte b) muestra a **`ptrTemp`** apuntando hacia el nodo eliminado de la cola, y a **`ptrCabeza`** apuntando al nuevo primer nodo de la cola. La función **`free`** se utiliza para solicitar la memoria apuntada por **`ptrTemp`**.

## 12.7 Árboles

Las listas ligadas, las pilas y las colas son *estructuras de datos lineales*. Un *árbol* es una estructura de datos no lineal de dos dimensiones, con propiedades especiales. Tres nodos contienen dos o más ligas. Esta sección explica los *árboles binarios* (figura 12.17); árboles cuyos nodos contienen dos ligas (ninguna, una, o ambas de las cuales pueden ser **`NULL`**). El *nodo raíz* es el primer nodo del árbol. Cada liga del nodo raíz hace referencia a un *hijo*. El *hijo izquierdo* es el primer nodo del *subárbol izquierdo*, y el *hijo derecho* es el primer nodo del

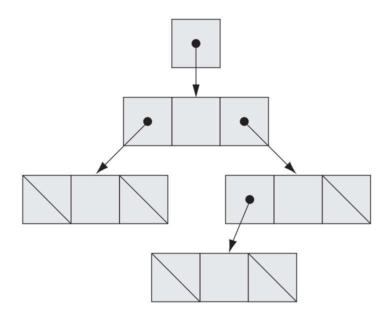


**Figura 12.17** Representación gráfica de un árbol binario.

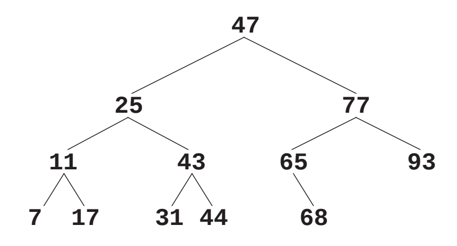



**Figura 12.18** Árbol binario de búsqueda.

*subárbol derecho*. A los hijos de un nodo se les conoce como *hermanos*. A un nodo sin hijos se le conoce como *nodo hoja*. Los científicos en computación generalmente dibujan árboles del nodo raíz hacia abajo; exactamente de manera contraria a los árboles naturales.

En esta sección creamos un árbol binario especial llamado *árbol binario de búsqueda*. Un árbol binario de búsqueda (sin valores duplicados de nodos) tiene la característica de que los valores de cualquier subárbol izquierdo son menores que el valor de su nodo padre, y que los valores de cualquier subárbol derecho son mayores que el valor de su *nodo padre*. La figura 12.18 muestra un árbol binario de búsqueda con 12 valores. Observe que la forma del árbol binario de búsqueda que corresponde al conjunto de datos puede variar, de acuerdo con el orden en que se inserten los valores en el árbol.

### Error común de programación 12.8

*No establecer en* **`NULL`** *las ligas de los nodos hoja de un árbol, puede ocasionar errores de ejecución.*

La figura 12.19, cuya salida aparece en la figura 12.20, crea un árbol binario de búsqueda y lo recorre de tres formas: *inorden*, en *preorden* y en *postorden*. El programa genera 10 números aleatorios e inserta cada uno en el árbol, con excepción de los valores duplicados.

```
/* Figura 12.19: fig12_19.c
   Crea un árbol binario y lo recorre en
   preorden, inorden, y en postorden */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* estructura autorreferenciada */
struct nodoArbol {
   struct nodoArbol *ptrIzq;  /* apuntador al subárbol izquierdo */
   int dato; /* valor del nodo */
   struct nodoArbol *prtDer; /* apuntador al subárbol derecho */
}; /* fin de la estructura nodoArbol */

typedef struct nodoArbol NodoArbol; /* sinónimo de la estructura nodoArbol */
typedef NodoArbol *ptrNodoArbol;    /* sinónimo de NodoArbol* */

/* prototipos */
void insertaNodo( ptrNodoArbol *ptrArbol, int valor );
void inOrden( ptrNodoArbol ptrArbol );
void preOrden( ptrNodoArbol ptrArbol );
void postOrden( ptrNodoArbol ptrArbol );

/* la función main comienza la ejecución del programa */
int main()
```

**Figura 12.19** Creación y recorrido de un árbol binario. (Parte 1 de 3.)

```
{
   int i; /* contador para el ciclo de 1 a 10 */
   int elemento; /* variable para almacenar valores al azar */
   ptrNodoArbol ptrRaiz = NULL; /* árbol inicialmente vacío */

   srand( time( NULL ) );
   printf( "Los numeros colocados en el arbol son:\n" );

   /* inserta valores al azar entre 1 y 15 en el árbol */
   for ( i = 1; i <= 10; i++ ) {
      elemento = rand() % 15;
      printf( "%3d", elemento );
      insertaNodo( &ptrRaiz, elemento );
   } /* fin de for */

   /* recorre el árbol en preorden */
   printf( "\n\nEl recorrido en preorden es:\n" );
   preOrden( ptrRaiz );

   /* recorre el árbol en in inorden */
   printf( "\n\nEl recorrido inorden es:\n" );
   inOrden( ptrRaiz );

   /* recorre el árbol en posorden */
   printf( "\n\nEl recorrido en posorden es:\n" );
   posOrden( ptrRaiz );

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* inserta un nodo dentro del árbol */
void insertaNodo( ptrNodoArbol *ptrArbol, int valor )
{

   /* si el árbol está vacío */
   if ( *ptrArbol == NULL ) {
      *ptrArbol = malloc( sizeof( NodoArbol ) );

      /* si la memoria está asignada, entonces asigna el dato */
      if ( *ptrArbol != NULL ) {
         ( *ptrArbol )->dato = valor;
         ( *ptrArbol )->ptrIzq = NULL;
         ( *ptrArbol )->prtDer = NULL;
      } /* fin de if */
      else {
         printf( "no se inserto %d. No hay memoria disponible.\n", valor );
      } /* fin de else */

   } /* fin de if */
   else { /* el árbol no está vacío */

      /* el dato a insertar es menor que el dato en el nodo actual */
      if ( valor < ( *ptrArbol )->dato ) {
         insertaNodo( &( ( *ptrArbol )->ptrIzq ), valor );
```

**Figura 12.19** Creación y recorrido de un árbol binario. (Parte 2 de 3.)

```
      } /* fin de if */

      /* el dato a insertar es mayor que el dato en el nodo actual */
      else if ( valor > ( *ptrArbol )->dato ) {
         insertaNodo( &( ( *ptrArbol )->prtDer ), valor );
      } /* fin de else if */
      else { /* ignora el valor duplicado del dato */
         printf( “dup” );
      } /* fin de else */

   } /* fin de else */

} /* fin de la función insertaNodo */

/* comienza el recorrido inorden del árbol */
void inOrden( ptrNodoArbol ptrArbol )
{

   /* si el árbol no está vacío, entonces recórrelo */
   if ( ptrArbol != NULL ) {
      inOrden( ptrArbol->ptrIzq );
      printf( “%3d”, ptrArbol->dato );
      inOrden( ptrArbol->prtDer );
   } /* fin de if */

} /* fin de la función inOrden */

/* comienza el recorrido preorden del árbol */
void preOrden( ptrNodoArbol ptrArbol )
{

   /* si el árbol no está vacío, entonces recórrelo */
   if ( ptrArbol != NULL ) {
      printf( “%3d”, ptrArbol->dato );
      preOrden( ptrArbol->ptrIzq );
      preOrden( ptrArbol->prtDer );
   } /* fin de if */

} /* fin de la función preOrden */

/* comienza el recorrido postorden del árbol */
void postOrden( ptrNodoArbol ptrArbol )
{

   /* si el árbol no está vacío, entonces recórrelo */
   if ( ptrArbol != NULL ) {
      postOrden( ptrArbol->ptrIzq );
      postOrden( ptrArbol->prtDer );
      printf( “%3d”, ptrArbol->dato );
   } /* fin de if */

} /* fin de la función posOrden */
```

**Figura 12.19** Creación y recorrido de un árbol binario. (Parte 3 de 3.)

```
Los numeros colocados en el arbol son:
  1  9 13  8  3  3dup  2  5  6  5dup

El recorrido en preOrden es:
  1  9  8  3  2  5  6 13

El recorrido inOrden es:
  1  2  3  5  6  8  9 13

El recorrido en posOrden es:
  2  6  5  3  8 13  9  1
```

**Figura 12.20** Salida de ejemplo del programa correspondiente a la figura 12.19.

Las funciones utilizadas en la figura 12.19 para crear y recorrer un árbol binario de búsqueda, son recursivas. La función **`insertaNodo`** (líneas 58 a 93) recibe como argumentos la dirección del árbol y un entero a almacenarse en el árbol. *En un árbol binario de búsqueda, sólo puede insertarse un nodo en la forma de nodo hoja*. Los pasos para insertar un nodo en un árbol binario de búsqueda son los siguientes:

1. Si **`*ptrArbol`** es **`NULL`** (línea 62), crea un nuevo nodo (línea 63). Llama a **`malloc`**, asigna a **`*ptrArbol`** la memoria asignada, asigna a **`(*ptrArbol)->dato`** el entero a almacenarse (línea 67), asigna el valor **`NULL`** a **`(*ptrArbol)->ptrIzq`** y a **`(*ptrArbol)->ptrDer`** (líneas 68 y 69), y devuelve el control a quien hizo la llamada (ya sea a **`main`** o a una llamada anterior a **`insertaNodo`**).
2. Si el valor de **`*ptrArbol`** no es **`NULL`** y el valor a insertar es menor que **`(*ptrArbol)->dato`**, la función **`insertaNodo`** es llamada con la dirección de **`(*ptrArbol)->ptrIzq`** (línea 80). Si el valor a insertar es mayor que **`(*ptrArbol)->dato`**, la función **`insertaNodo`** es llamada con la dirección de **`(*ptrArbol)->ptrDer`** (línea 85). De lo contrario, los pasos recursivos continúan hasta que se encuentra un apuntador **`NULL`**, después se ejecuta el paso 1) para insertar el nuevo nodo.

Las funciones **`inOrden`** (líneas 96 a 106), **`preOrden`** (líneas 109 a 119) y **`postOrden`** (líneas 122 a 132) reciben un árbol (es decir, el apuntador hacia el nodo raíz del árbol), y lo recorren.

Los pasos para un recorrido **`inOrden`** son:

1. Recorre el subárbol izquierdo **`inOrden`**.
2. Procesa el valor del nodo.
3. Recorre el subárbol derecho **`inOrden`**.

El valor de un nodo no se procesa hasta que se procesan los valores de su subárbol izquierdo. El recorrido **`inOrden`** del árbol correspondiente a la figura 12.21 es:

Observe que el recorrido **`inOrden`** de un árbol binario de búsqueda imprime los valores de los nodos en orden ascendente. El proceso de creación de un árbol binario de búsqueda en realidad ordena los datos y, por lo tanto, a este proceso se le conoce como *ordenamiento de un árbol binario*.

Los pasos para un recorrido en **`preOrden`** son:

1. Procesa el valor del nodo.
2. Recorre el subárbol izquierdo en **`preOrden`**.

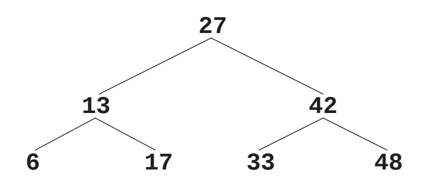



**Figura 12.21** Árbol binario de búsqueda con siete nodos.

3. Recorre el subárbol derecho en **preOrden**.

El valor de cada nodo se procesa conforme se visitan los nodos. Después de que se procesa el valor de un nodo dado, se procesan los valores del subárbol izquierdo, y después los valores del subárbol derecho. El recorrido en **preOrden** del árbol correspondiente a la figura 12.21 es:

```
27 13 6 17 42 33 48
```

Los pasos para un recorrido en **postOrden** son:

1. Recorre el subárbol izquierdo en **postOrden**.
2. Recorre el subárbol derecho en **postOrden**.
3. Procesa el valor del nodo.

El valor de cada nodo no se imprime hasta que se imprimen los valores de sus hijos. El recorrido en **postOrden** del árbol correspondiente a la figura 12.21 es:

```
6 17 13 33 48 42 27
```

El árbol binario de búsqueda facilita la eliminación de duplicados. Conforme se crea el árbol, se reconoce cualquier intento de insertar un valor duplicado, ya que en cada comparación, un duplicado seguirá las mismas decisiones, "ve hacia la izquierda" o "ve hacia la derecha", que el valor original. Por lo tanto, el duplicado en algún momento se comparará con un nodo del árbol que contenga el mismo valor. En ese momento, el valor duplicado simplemente se descarta.

Buscar en un árbol binario un elemento que coincida con un valor clave también es rápido. Si el árbol se compacta lo suficiente, cada nivel contendrá alrededor del doble de los elementos que el nivel anterior. Por lo tanto, un árbol binario de búsqueda con *n* elementos tendría un máximo de $\log_{2n}$ niveles, y tendría que hacer un máximo de $\log_{2n}$ comparaciones para encontrar una coincidencia, o para determinar que no existe alguna. Esto significa, por ejemplo, que cuando se hace una búsqueda en un árbol binario de búsqueda (muy compactado) de 1000 elementos, no se necesitan más de 10 comparaciones, ya que $2^{10} > 1000$. Cuando se hace una búsqueda en un árbol binario de búsqueda (muy compactado) de 1,000,000 elementos, no se necesitan más de 20 comparaciones, ya que $2^{20} > 1,000,000$.

En los ejercicios, presentamos algoritmos para otras operaciones con árboles binarios, como eliminar un elemento del árbol, impresión un árbol binario en un formato bidimensional, y realizar un recorrido del árbol en orden de niveles. El recorrido en orden de niveles visita los nodos del árbol fila por fila, comenzando en el nivel del nodo raíz. En cada nivel del árbol, los nodos son visitados de izquierda a derecha. Otros ejercicios con árboles binarios incluyen que un árbol binario de búsqueda pueda contener valores duplicados; la inserción de valores de tipo cadena en un árbol; y determinar cuántos niveles hay en un árbol binario.

## RESUMEN

- Las estructuras autorreferenciadas contienen miembros llamados ligas que apuntan a estructuras del mismo tipo.
- Las estructuras autorreferenciadas permiten que muchas estructuras estén ligadas en pilas, colas, listas y árboles.
- La asignación dinámica de memoria reserva un bloque de bytes en memoria para almacenar un objeto de datos durante la ejecución de un programa.
- La función **malloc** toma como argumento el número de bytes a asignar, y devuelve un apuntador **void** hacia la memoria asignada. Por lo general, la función **malloc** se utiliza con el operador **sizeof**. El operador **sizeof** determina el tamaño en bytes de la estructura a la que se le está asignando memoria.
- La función **free** libera memoria.
- Una lista ligada es una colección de datos almacenado en un grupo de estructuras autorreferenciadas conectadas.
- Una lista ligada es una estructura de datos dinámica; la longitud de la lista puede aumentar o disminuir, conforme sea necesario.
- Las listas ligadas pueden continuar creciendo, mientras exista memoria disponible.
- Las listas ligadas proporcionan un mecanismo para la inserción y la eliminación simple de datos, mediante la reasignación de apuntadores.

- Las pilas y las colas son versiones restringidas de una lista ligada.
- Los nuevos nodos se agregan a una pila y se eliminan de ella, sólo en la cima. Por esta razón, a las pilas se les conoce como estructuras de datos últimas en entrar, primeras en salir (UEPS).
- El miembro liga del último nodo de la pila se establece en **`NULL`** para indicar el fondo de la pila.
- Las dos operaciones básicas que se utilizan para manipular una pila son **`empujar`** (push) y **`sacar`** (pop). La operación **`empujar`** crea un nuevo nodo y lo coloca en la cima de la pila. La operación **`sacar`** elimina un nodo de la cima de la pila, libera la memoria que estaba asignada al nodo eliminado y devuelve el valor eliminado.
- En una cola, los nodos se eliminan de la cabeza y se agregan en el talón. Por esta razón, a la cola se le conoce como estructura de datos primera en entrar, primera en salir (PEPS). Las operaciones para agregar y eliminar se conocen como **`agregar`** (enqueue) y **`retirar`** (dequeue).
- Los árboles son estructuras de datos más complejas que las listas ligadas, las colas y las pilas. Los árboles son estructuras de datos bidimensionales que requieren dos o más ligas por nodo.
- Los árboles binarios contienen dos ligas por nodo.
- El nodo raíz es el primer nodo del árbol.
- Cada uno de los apuntadores del nodo raíz hace referencia a un hijo. El hijo izquierdo es el primer nodo del subárbol izquierdo y el hijo derecho es el primer nodo del subárbol derecho. A los hijos de un nodo se les conoce como hermanos. Si un nodo no tiene hijos, a éste se le llama nodo hoja.
- Un árbol binario de búsqueda tiene la característica de que el valor del hijo izquierdo de un nodo es menor que el valor del nodo padre, y el valor del hijo derecho de un nodo es mayor o igual que el valor del nodo padre. Si puede determinarse que no hay valores duplicados, el valor del hijo derecho es simplemente mayor que el valor del nodo padre.
- Un recorrido inorden de un árbol binario recorre el subárbol izquierdo inorden, procesa el valor del nodo y recorre el subárbol derecho inorden. El valor de un nodo no se procesa hasta que los valores de su subárbol izquierdo se procesan.
- Un recorrido en preorden procesa el valor del nodo, recorre el subárbol izquierdo en preorden y recorre el subárbol derecho en preorden. El valor de cada nodo se procesa, conforma se encuentra cada nodo.
- Un recorrido en postorden recorre el subárbol izquierdo en postorden, recorre el subárbol derecho en postorden, y procesa el valor del nodo. El valor de cada nodo no se procesa hasta que los valores de ambos subárboles se procesan.

## *TERMINOLOGÍA*

**`agregar (enqueue)`**
apuntador a un apuntador
apuntador **`NULL`**
árbol
árbol binario
árbol binario de búsqueda
asignación dinámica de memoria
cabeza de una cola
cima
cola
doble indirección
eliminación de un nodo
**`empujar (push)`**
estructura autorreferenciada
estructura de datos lineal
estructura de datos no lineal
estructuras de datos dinámicas
**`free`**
función predicado
hermanos
hijo derecho
hijo izquierdo
hijos
inserción de un nodo
lista ligada
**`malloc`** (asigna memoria)
nodo
nodo hijo
nodo hoja
nodo padre
nodo raíz
ordenamiento de un árbol binario
PEPS (primera en entrar, primera en salir)
pila
recorrido
recorrido en postorden
recorrido en preorden
recorrido inorden
**`retirar (dequeue)`**
**`sacar (pop)`**
**`sizeof`**
subárbol
subárbol derecho
subárbol izquierdo
talón de una cola
UEPS (última en entrar, primera en salir)
visita a un nodo

## *ERRORES COMUNES DE PROGRAMACIÓN*

**12.1** No establecer en **`NULL`** la liga del último nodo de una lista, puede provocar errores de ejecución.

**12.2** Suponer que el tamaño de una estructura es la suma del tamaño de sus miembros, es un error lógico.

**12.3** No devolver la memoria asignada dinámicamente cuando ya no es necesaria, puede ocasionar que el sistema se quede sin memoria de manera prematura. A esto se le conoce en ocasiones como “fuga de memoria”.

**12.4** Liberar memoria no asignada dinámicamente con **`malloc`**, es un error.

**12.5** Hacer referencia a memoria que ha sido liberada, es un error.

**12.6** No establecer en **`NULL`** la liga del nodo del fondo de una pila puede ocasionar errores de ejecución.

**12.7** No establecer en **`NULL`** la liga del último nodo de una cola, puede ocasionar errores de ejecución.

**12.8** No establecer en **`NULL`** las ligas de los nodos hoja de un árbol, puede ocasionar errores de ejecución.

## *TIPS PARA PREVENIR ERRORES*

**12.1** Cuando utilice **`malloc`**, evalúe la devolución de un valor de apuntador **`NULL`**. Imprima un mensaje de error si la memoria requerida no es asignada.

**12.2** Asigne **`NULL`** al miembro liga de un nuevo nodo. Los apuntadores deben inicializarse antes de que se utilicen.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**12.1** Utilice el operador **`sizeof`** para determinar el tamaño de una estructura.

**12.2** Cuando la memoria que se asignó dinámicamente ya no es necesaria, utilice **`free`** para devolverla inmediatamente al sistema.

## *TIPS DE RENDIMIENTO*

**12.1** Un arreglo puede declararse para que contenga más elementos que los esperados, sin embargo, esto puede desperdiciar memoria. Las listas ligadas proporcionan una mejor utilización de memoria, en estas situaciones.

**12.2** Las inserciones y las eliminaciones en un arreglo ordenado pueden llevar demasiado tiempo; todos los elementos que siguen al elemento insertado o eliminado deben desplazarse adecuadamente.

**12.3** Los elementos de un arreglo se almacenan en memoria de manera contigua. Esto permite el acceso inmediato a un elemento de un arreglo, ya que la dirección de cualquier elemento puede calcularse directamente de acuerdo con su posición relativa al principio del arreglo. Las listas ligadas no permiten el acceso inmediato a sus elementos.

**12.4** Utilizar la asignación dinámica de memoria (en lugar de arreglos) para estructuras de datos que aumentan y disminuyen en tiempo de ejecución, puede ahorrar memoria. Sin embargo, recuerde que los apuntadores ocupan más espacio, y que la asignación dinámica de memoria incurre en la sobrecarga de llamadas a funciones.

## *TIP DE PORTABILIDAD*

**12.1** El tamaño de una estructura no necesariamente es la suma de los tamaños de sus miembros. Esto es así debido a los diversos requerimientos de los límites de alineación que dependen de cada máquina (vea el capítulo 10).

## *EJERCICIOS DE AUTOEVALUACIÓN*

**12.1** Complete los espacios en blanco:

a) Una estructura auto____________ se utiliza para formar estructuras de datos dinámicas.
b) La función ____________ se utiliza para asignar memoria dinámicamente.
c) Una ____________ es una versión especializada de una lista ligada, en la que los nodos pueden insertarse y eliminarse sólo del inicio de la lista.
d) Las funciones que revisan una lista ligada, pero que no la modifican se conocen como ____________.
e) Una cola se conoce como una estructura de dato ____________.
f) El apuntador al siguiente nodo de una lista ligada se conoce como una ____________.
g) La función ____________ se utiliza para solicitar la memoria asignada dinámicamente.
h) Una ____________ es una versión especializada de una lista ligada, en la que los nodos pueden insertarse sólo al inicio de la lista, y eliminarse sólo al final de la lista.
i) Un ____________ es una estructura de datos no lineal de dos dimensiones que contiene nodos con dos o más ligas.
j) A una pila se le conoce como una estructura de datos ____________, ya que el último nodo que se inserta es el primer nodo eliminado.

k) Los nodos de un árbol ___________ contienen dos miembros ligados.
l) El primer nodo de un árbol es el nodo ___________.
m) Cada liga de un nodo de un árbol apunta hacia un ___________ o hacia un ___________ de ese árbol.
n) El nodo de un árbol que no tiene hijos se conoce como nodo ___________.
o) Los algoritmos para recorrer un árbol (que tratamos en este capítulo) binario son ___________, ___________ y ___________.

**12.2** ¿Cuáles son las diferencias entre una lista ligada y una pila?

**12.3** ¿Cuáles son las diferencias entre una pila y una cola?

**12.4** Escriba una instrucción o un conjunto de instrucciones para realizar las siguientes tareas. Suponga que todas las manipulaciones ocurren en **main** (por lo tanto, ninguna dirección de variables apuntador es necesaria), y suponga las siguientes definiciones:

```
struct nodoCalificacion {
   char apellido[ 20 ];
   double calificacion;
   struct nodoCalificacion *ptrSiguiente;
};
typedef struct nodoCalificacion NodoCalificacion;
typedef NodoCalificacion *ptrNodoCalificacion;
```

a) Cree un apuntador hacia el inicio de la lista llamado **ptrInicio**. La lista está vacía.
b) Cree un nuevo nodo de tipo **NodoCalificacion** que es apuntado por el apuntador **ptrNuevo** del tipo **ptrNodoCalificacion**. Asigne la cadena **“Perez”** al miembro **apellido** y el valor **91.5** al miembro **calificacion** (utilice **strcpy**). Proporcione cualquier declaración e instrucción necesaria.
c) Suponga que la lista apuntada por **ptrInicio** actualmente consta de dos nodos; uno que contiene **“Perez”** y otro que contiene **“Sanchez”**. Los nodos están en orden alfabético. Proporcione las instrucciones necesarias para insertar nodos en orden, que contengan los siguientes datos para **apellido** y **calificacion**:

```
“Fernandez”   85.0
“Lopez”       73.5
“Martinez”    66.5
```

Utilice los apuntadores **ptrAnterior**, **ptrActual** y **ptrNuevo** para realizar las inserciones. Establezca a qué apuntan **ptrAnterior** y **ptrActual** antes de cada inserción. Suponga que **ptrNuevo** siempre apunta al nuevo nodo, y que el dato ya se asignó al nuevo nodo.
d) Escriba un ciclo **while** que imprima los datos de cada nodo de la lista. Utilice el apuntador **ptrActual** para moverse a lo largo de la lista.
e) Escriba un ciclo **while** que elimine todos los nodos de la lista y que libere la memoria asociada con cada nodo. Utilice el apuntador **ptrActual** y el apuntador **ptrTemp** para recorrer la lista y liberar memoria, respectivamente.

**12.5** Manualmente proporcione los recorridos **inOrden**, en **preOrder** y en **postOrden** del árbol binario de búsqueda de la figura 12.22.

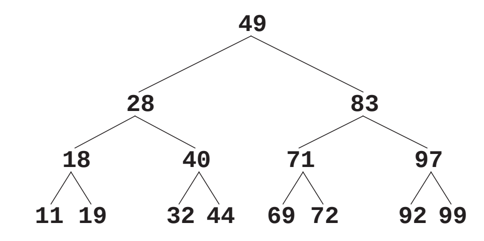



**Figura 12.22** Un árbol binario de búsqueda con 15 nodos.

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**12.1** a) Referenciada. b) **malloc**. c) Pila. d) Predicado. e) PEPS. f) Liga. g) **free**. h) Cola. i) Árbol. j) UEPS. k) Binario. l) Raíz. m) Hijo. n) Hoja. o) Inorden, preorden, postorden.

**12.2** Es posible insertar y eliminar un nodo en cualquier parte de una lista ligada. Sin embargo, los nodos de una pila sólo pueden insertarse y eliminarse en la cima de la pila.

**12.3** Una cola tiene apuntadores tanto a su cabeza como a su talón, por lo que los nodos pueden insertarse en el talón y eliminarse de la cabeza. Una pila tiene un solo apuntador a la cima, en donde se realizan las inserciones y las eliminaciones de los nodos.

**12.4** a) `ptrNodoCalificacion ptrInicio = NULL;`

b)
```
ptrNodoCalificacion ptrNuevo;
ptrNuevo = malloc( sizeof( NodoCalificacion ) );
strcpy( ptrNuevo->apellido, "Perez" );
ptrNuevo->calificacion = 91.5;
ptrNuevo->ptrSiguiente = NULL;
```

c) Para insertar **“Fernandez”**:

**ptrAnterior** es **NULL**, **ptrActual** apunta al primer elemento de la lista.
```
ptrNuevo->ptrSiguiente = ptrActual;
ptrInicio = ptrNuevo;
```

Para insertar **“Lopez”**:

**ptrAnterior** apunta al último elemento de la lista (que contiene **“Sanchez”**)
**ptrActual** es **NULL**.
```
ptrNuevo->ptrSiguiente = ptrActual;
ptrAnterior->ptrSiguiente = ptrNuevo;
```

Para insertar **“Martinez”**:

**ptrAnterior** apunta al nodo que contiene **“Perez”**
**ptrActual** apunta al nodo que contiene **“Sanchez”**
```
ptrNuevo->ptrSiguiente = ptrActual;
ptrAnterior->ptrSiguiente = ptrNuevo;
```

d)
```
ptrActual = ptrInicio;
while( ptrActual != NULL ) {
   printf( "Apellido = %s\nCalificacion = %6.2f\n",
     ptrActual->apellido, ptrActual->calificacion );
   ptrActual = ptrActual->ptrSiguiente;
}
```

e)
```
ptrActual = ptrInicio;
while( ptrActual != NULL ) {
   ptrTemp = ptrActual;
   ptrActual = ptrActual->ptrSiguiente;
   free( ptrTemp );
}
ptrInicio = NULL;
```

**12.5** El recorrido **inOrden** es:

**11 18 19 28 32 40 44 49 69 71 72 83 92 97 99**

El recorrido en **preOrden** es:

**49 28 18 11 19 40 32 44 83 71 69 72 97 92 99**

El recorrido en **postOrden** es:

**11 19 18 32 44 40 28 69 72 71 92 99 97 83 49**

## *EJERCICIOS*

**12.6** Escriba un programa que concatene dos listas ligadas de caracteres. El programa debe incluir la función concatenar que tome como argumentos apuntadores a ambas listas, y que concatene la segunda lista a la primera.

**12.7** Escriba un programa que mezcle dos listas ordenadas de enteros en una sola lista ordenada de enteros. La función mezclar debe recibir apuntadores al primer nodo de cada lista a mezclar, y debe devolver un apuntador al primer nodo de la lista mezclada.

**12.8** Escriba un programa que inserte en orden 25 enteros al azar, del 0 al 100, en una lista ligada. El programa debe calcular la suma de los elementos y el promedio en punto flotante de ellos.

**12.9** Escriba un programa que genere una lista ligada de 10 caracteres, y que después genere una copia de la lista en orden inverso.

**12.10** Escriba un programa que introduzca una línea de texto, y que después utilice una pila para imprimir dicha línea en orden inverso.

**12.11** Escriba un programa que utilice una pila para determinar si una cadena es un palíndromo (es decir, que la cadena diga exactamente lo mismo si se lee hacia adelante o hacia atrás). El programa debe ignorar los espacios y la puntuación.

**12.12** Los compiladores utilizan las pilas para ayudar en el proceso de evaluación de expresiones y en la generación de código en lenguaje máquina. En éste y en el siguiente ejercicio, investigaremos cómo es que los compiladores evalúan expresiones aritméticas que sólo constan de constantes, operadores y paréntesis.

Los humanos por lo general escriben expresiones como **`3 + 4`** y **`7 / 9`**, en las que el operador **`+`** o **`/`** (en este caso) se escriben entre los operandos; a esto se le conoce como *notación infijo*. Las computadoras "prefieren" la *notación postfijo*, en la que el operador se escribe a la derecha de sus dos operandos. Las expresiones infijo anteriores aparecerían en notación postfijo como **`3 4 +`** y **`7 9 /`**, respectivamente.

Para evaluar una expresión infijo compleja, un compilador primero convertiría la expresión a notación postfijo, y evaluaría ésta versión de la expresión. Cada uno de estos algoritmos requiere sólo una pasada de la expresión de izquierda a derecha. Cada algoritmo utiliza una pila para dar soporte a su operación, y en cada uno se utiliza una pila para un propósito diferente.

En este ejercicio, usted escribirá una versión del algoritmo de conversión de infijo a postfijo. En el siguiente, usted escribirá una versión del algoritmo de evaluación de la expresión postfijo.

Escriba un programa que convierta una expresión aritmética ordinaria en notación infijo con enteros de un solo dígito como la siguiente (suponga que se introduce una expresión válida)

```
(6 + 2) * 5 – 8 / 4
```

a una expresión postfijo. La versión postfijo de la expresión infijo anterior es

```
6 2 + 5 * 8 4 / -
```

El programa debe leer la expresión en un arreglo de caracteres llamado **`infijo`**, y utilizar las versiones modificadas de las funciones pila implementadas en este capítulo, para ayudar a generar la expresión postfijo en el arreglo de caracteres llamado **`postfijo`**. El algoritmo para crear una expresión postfijo es el siguiente:

1) Meter un paréntesis izquierdo **`'('`** en la pila.
2) Agregar un paréntesis derecho **`')'`** al final de infijo.
3) Mientras la pila no esté vacía, leer **`infijo`** de izquierda a derecha y hacer lo siguiente:
    - Si el carácter actual en **`infijo`** es un dígito, cópialo al siguiente elemento de **`postfijo`**.
    - Si el carácter actual en **`infijo`** es un paréntesis izquierdo, mételo en la pila.
    - Si el carácter actual en **`infijo`** es un operador,
        - Saca los operadores (si hay alguno) de la cima de la pila, mientras tengan una precedencia mayor o igual que la del operador actual, e inserta en **`postfijo`** los operadores sacados.
        - Mete el carácter actual de **`infijo`** en la pila.
    - Si el carácter actual en **`infijo`** es un paréntesis derecho
        - Saca los operadores de la cima de la pila e insértalos en **`postfijo`**, hasta que haya un paréntesis izquierdo en la cima de la pila.
        - Saca (y descarta) el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

+ suma
- resta
* multiplicación
/ división
^ exponenciación
% módulo

La pila debe mantenerse con las siguientes declaraciones:

```
struct nodoPila {
   char dato;
   struct nodoPila *ptrSiguiente;
};
typedef struct nodoPila NodoPila;
typedef NodoPila *ptrNodoPila;
```

El programa debe constar de una función **`main`** y otras ocho funciones con los siguientes encabezados de función:

```
void convierteAPostfijo( char infijo[ ], char postfijo[ ] )
```

Convierte la expresión infijo en notación postfijo.

```
int esOperador( char c )
```

Determina si **`c`** es un operador.

```
int precedencia( char operador1, char operador2 )
```

Determina si la precedencia del **`operador1`** es menor, igual o mayor que la precedencia del **`operador2`**. La función devuelve **`-1`**, **`0`** y **`1`**, respectivamente.

```
void empujar( ptrNodoPila *ptrCima, char valor )
```

Mete un valor a la pila.

```
char sacar( ptrNodoPila *ptrCima)
```

Saca un valor de la pila.

```
char cimaPila( ptrNodoPila ptrCima)
```

Devuelve el valor en la cima de la pila, sin sacarlo de ella.

```
int estaVacia( ptrNodoPila ptrCima )
```

Determina si la pila está vacía.

```
void imprimePila( ptrNodoPila ptrCima )
```

Imprime la pila.

**12.13** Escriba un programa que evalúe una expresión postfijo (suponga que es válida) como:

```
6 2 + 5 * 8 4 / -
```

El programa debe leer una expresión postfijo que conste de dígitos y operadores en un arreglo de caracteres. Por medio de versiones modificadas de funciones pila implementadas en este capítulo, el programa debe explorar la expresión y evaluarla. El algoritmo es el siguiente:

1) Agregar el carácter nulo (**`'\0'`**) al final de la expresión postfijo. Cuando se encuentre el carácter nulo, no se necesitará mayor procesamiento.
2) Mientras no se encuentre el **`'\0'`**, lee la expresión de izquierda a derecha.
    Si el carácter actual es un dígito,
        Mete su valor entero en la pila (el valor entero de un dígito carácter es su valor en el conjunto de caracteres de la computadora, menos el valor de **`'\0'`** en el conjunto de caracteres de la computadora).
    De lo contrario, si el carácter actual es un *operador*,
        Saca los dos elementos de la cima de la pila hacia las variables **`x`** y **`y`**.
        Calcula **`y`** *operador* **`x`**.
        Mete el resultado del cálculo en la pila.
3) Cuando se encuentre el carácter nulo en la expresión, saca el valor de la cima de la pila. Éste es el resultado de la expresión postfijo.

[*Nota:* En el paso 2) anterior, si el operador es **`'\'`**, la cima de la pila es **`2`**, y el siguiente elemento de la pila es **`8`**, después saca **`2`** hacia **`x`**, saca **`8`** hacia **`y`**, evalúa **`8 / 2`**, y mete el resultado, **`4`**, de regreso a la pila. Esta nota también aplica para el operador **`'-'`**.] Las operaciones aritméticas permitidas en una expresión son:

+ suma
− resta
* multiplicación
/ división
^ exponenciación
% módulo]

La pila debe mantenerse con las siguientes declaraciones:

```
struct nodoPila {
   int dato;
   struct nodoPila *ptrSiguiente;
};

typedef struct nodoPiloa NodoPila;
typedef NodoPila *ptrNodoPila;
```

El programa debe constar de una función **main** y otras seis instrucciones con los siguientes encabezados de función:

```
int evaluaExpresionPostfijo( char *expr )
```

Evalúa la expresión postfijo.

```
int calcula( int op1, int op2, char operador )
```

Evalúa la expresión **op1** operador **op2**.

```
void empujar( ptrNodoPila *ptrCima, int valor )
```

Mete un valor a la pila.

```
int sacar( ptrNodoPila *ptrCima )
```

Saca un valor de la pila.

```
int estaVacia( ptrNodoPila ptrCima )
```

Determina si la pila está vacía.

```
void imprimePila( ptrNodoPila ptrCima )
```

Imprime la pila.

**12.14** Modifique el programa evaluador de expresiones postfijo correspondiente al ejercicio 12.13, para que pueda procesar operandos enteros mayores que **9**.

**12.15** (*Simulación de un supermercado.*) Escriba un programa que simule una fila para pagar en un supermercado. La fila es una cola. Los clientes llegan en intervalos enteros aleatorios de 1 a 4 minutos. Obviamente, el flujo de llegada debe estar equilibrado. Si el promedio del flujo de llegada es mayor que el flujo promedio de servicio, la cola crecerá infinitamente. Incluso con flujos equilibrados, la aleatoriedad puede ocasionar filas largas. Ejecute la simulación del supermercado para 12 horas diarias (720 minutos), por medio del siguiente algoritmo.

1) Elija un entero al azar entre 1 y 4 para determinar el minuto en el que llegó el primer cliente.
2) En el tiempo de llegada del primer cliente:
   Determine el tiempo de atención al cliente (un entero al azar entre 1 y 4);
   Comience a atender al cliente;
   Programe el tiempo de llegada del siguiente cliente (un entero al azar entre 1 y 4, sumado al tiempo actual).
3) Para cada minuto del día:
   - Si el siguiente cliente llega,
     - Decirlo así;
     - Coloque al cliente en la cola;
     - Programe el tiempo de llegada del siguiente cliente;
   - Si la atención concluyó para el último cliente;
     - Decirlo así;
     - Saque de la cola al siguiente cliente que atenderá;
     - Determine el tiempo en el que se concluyó la atención al cliente (un entero al azar entre 1 y 4, sumado al tiempo actual).

Ahora ejecute su simulación para 720 minutos, y responda las siguientes preguntas:

a) ¿Cuál es el máximo número de clientes en la cola, en cualquier momento?
b) ¿Cuál es la espera más larga que un cliente experimenta?
c) ¿Qué ocurre si el intervalo de llegada se modifica de 1 a 4 minutos a 1 a 3 minutos?

**12.16** Modifique el programa de la figura 12.19 para permitir que el árbol binario contenga valores duplicados.

**12.17** Escriba un programa basado en el programa de la figura 12.19 que introduzca una línea de texto, que separe en tokens un enunciado, que inserte las palabras en un árbol binario de búsqueda, y que imprima los recorridos inorden, en preorden y en postorden del árbol.

[*Pista:* Lea la línea de texto en un arreglo. Utilice **`strtok`** para separar en tokens el texto. Cuando se encuentre un token, genere un nuevo nodo para el árbol, asigne el apuntador devuelto por **`strtok`** al miembro cadena del nuevo nodo, e inserte el nodo en el árbol.]

**12.18** En este capítulo, vimos que la eliminación de duplicados es directa, cuando se crea un árbol binario de búsqueda. Describa cómo realizaría una eliminación de duplicados, usando solamente un arreglo con un solo subíndice. Compare el rendimiento de la eliminación de duplicados basada en arreglos, con el rendimiento de la eliminación de duplicados basada en árboles binarios de búsqueda.

**12.19** Escriba una función llamada profundo, que reciba un árbol binario y que determine cuántos niveles tiene.

**12.20** (*Impresión recursiva de una lista en orden inverso.*) Escriba una función **`imprimeListaInversa`**, que recursivamente despliegue los elementos de una lista en orden inverso. Utilice su función en un programa de prueba que genere una lista ordenada de enteros y que imprima la lista en orden inverso.

**12.21** (*Búsqueda recursiva en una lista.*) Escriba una función **`buscaLista`** que recursivamente busque un valor en una lista ligada. La función debe devolver un apuntador hacia el valor, si es que lo encuentra; de lo contrario, debe devolver **`NULL`**. Utilice su función en un programa de prueba que genere una lista de enteros. El programa debe indicar al usuario que introduzca un valor a localizar en la lista.

**12.22** (*Eliminación en un árbol binario.*) En este ejercicio, explicamos la eliminación de elementos de árboles binarios de búsqueda. El algoritmo de eliminación no es tan directo como el de inserción. Existen tres casos que podemos encontrar cuando eliminamos un elemento: el elemento se encuentra en un nodo hoja (es decir, no tiene hijos); el elemento se encuentra en un nodo que tiene un solo hijo; o el elemento se encuentra en un nodo que tiene dos hijos.

Si el elemento a eliminar se encuentra en un nodo hoja, el nodo se elimina y el apuntador del nodo padre se establece en **`NULL`**.

Si el elemento a eliminar se encuentra en un nodo con un hijo, el apuntador del nodo padre se establece para que apunte al nodo hijo, y el nodo que contiene el dato se elimina. Esto ocasiona que el nodo hijo tome el lugar del nodo eliminado del árbol.

El último caso es el más difícil. Cuando se elimina un nodo con dos hijos, otro nodo debe ocupar su lugar. Sin embargo, el apuntador del nodo padre no puede simplemente asignarse para que apunte a uno de los hijos del nodo a eliminar. En la mayoría de los casos, el árbol binario de búsqueda resultante no se apegará a las siguientes características de los árboles binarios de búsqueda: *los valores de cualquier subárbol izquierdo son menores que el valor del nodo padre, y los valores de cualquier subárbol derecho son mayores que el valor del nodo padre*.

¿Qué nodo se utiliza como nodo de reemplazo para mantener estas características? Ya sea el nodo que contenga el valor más grande del árbol, que sea menor que el valor del nodo que se está eliminando, o el nodo que contenga el valor más pequeño del árbol, que sea mayor que el valor del nodo que se está eliminando. Consideremos el nodo con el valor más pequeño. En un árbol binario de búsqueda, el valor más grande, menor que el valor de un nodo padre se localiza en el subárbol izquierdo de éste, y se garantiza que se encuentre en el nodo más a la derecha del subárbol. Este nodo se localiza recorriendo hacia la derecha el subárbol izquierdo, hasta que el apuntador hacia el hijo derecho del nodo actual sea **`NULL`**. Ahora estamos apuntando hacia el nodo de reemplazo, el cual es un nodo hoja o un nodo con un solo hijo a su izquierda. Si el nodo de reemplazo es un nodo hoja, los pasos para realizar la eliminación son los siguientes:

1) Almacenar el apuntador hacia el nodo a eliminar, en una variable apuntador temporal (este apuntador se utiliza para eliminar la memoria asignada dinámicamente).
2) Establecer el apuntador del padre del nodo a eliminar, para que apunte hacia el nodo de reemplazo.
3) Establezca en nulo al apuntador del padre del nodo de reemplazo.
4) Establecer el apuntador hacia el subárbol derecho del nodo de reemplazo, para que apunte hacia el subárbol derecho del nodo a eliminar.
5) Eliminar el nodo al que apunta la variable apuntador temporal.

Los pasos para la eliminación de un nodo de reemplazo con un hijo izquierdo son similares a los correspondientes a los nodos de reemplazo sin hijos, pero el algoritmo también debe mover el hijo hacia la posición del nodo de reemplazo. Si el nodo de reemplazo es uno con un hijo izquierdo, los pasos para realizar la eliminación son los siguientes:

1) Almacenar el apuntador hacia el nodo a eliminar, en una variable apuntador temporal.
2) Establecer el apuntador del padre del nodo a eliminar, para que apunte hacia el nodo de reemplazo.
3) Establecer el apuntador del padre del nodo de reemplazo, para que apunte hacia el hijo izquierdo del nodo de reemplazo.

4) Establecer el apuntador del subárbol derecho del nodo de reemplazo, para que apunte hacia el subárbol derecho del nodo a eliminar.
5) Eliminar el nodo al que apunta la variable apuntador temporal.

Escriba una función **`eliminarNodo`** que tome como argumentos el apuntador hacia el nodo raíz del árbol y el valor a eliminar. La función debe localizar en el árbol el nodo que contenga el valor a eliminar, y utilizar los algoritmos que explicamos aquí para eliminar el nodo. Si el valor no se encuentra en el árbol, la función debe imprimir un mensaje que indique si se eliminó o no el valor. Modifique el programa de la figura 12.19 para utilizar esta función. Después de eliminar un elemento, llame a las funciones de recorrido **`inorden`**, **`preorden`** y **`postorden`** para confirmar que la operación de eliminación se llevó a cabo correctamente.

**12.23** (*Búsqueda en un árbol binario*.) Escriba una función **`busquedaArbolBinario`** que intente localizar un valor especificado en un árbol binario de búsqueda. La función debe tomar como argumentos un apuntador al nodo raíz del árbol binario y una clave de búsqueda a localizar. Si se encuentra el nodo con la clave de búsqueda, la función debe devolver un apuntador hacia ese nodo; de lo contrario, la función debe devolver un apuntador **`NULL`**.

**12.24** (*Recorrido de un árbol binario en orden de niveles*.) El programa de la figura 12.19 mostró tres métodos para recorrer un árbol binario: inorden, en preorden y en postorden. Este ejercicio presenta el *recorrido en orden de niveles* de un árbol binario, en el que los valores de los nodos se imprimen nivel por nivel, comenzando en el nivel del nodo raíz. Los nodos de cada nivel se imprimen de izquierda a derecha. El recorrido en orden de niveles no es un algoritmo recursivo. Éste utiliza la estructura de datos cola para controlar la salida de los nodos. El algoritmo es el siguiente:

1) Insertar en la cola el nodo raíz.
2) Mientras haya nodos a la izquierda de la cola,
    Obtener el siguiente nodo de la cola
    Imprimir el valor del nodo
    Si el apuntador hacia el hijo izquierdo del nodo no es nulo
        Insertar en la cola el nodo hijo izquierdo
    Si el apuntador hacia el hijo derecho del nodo no es nulo
        Insertar en la cola el nodo hijo derecho.

Escriba una función **`ordenNiveles`** para realizar un recorrido en orden de niveles de un árbol binario. La función debe tomar como un argumento un apuntador hacia el nodo raíz del árbol binario. Modifique el programa de la figura 12.19 para utilizar esta función. Compare la salida de esta función con las salidas de los otros algoritmos de recorrido, para ver si éste funciona correctamente. [*Nota:* En este programa también necesitará modificar e incorporar las funciones para procesamiento de colas de la figura 12.13.]

**12.25** (*Impresión de árboles*.) Escriba una función recursiva **`salidaArbol`** para desplegar en la pantalla un árbol binario. La función debe desplegar el árbol fila por fila, con la cima del árbol a la izquierda de la pantalla, y el fondo del árbol hacia adelante a la derecha de la pantalla. Cada fila se despliega verticalmente. Por ejemplo, el árbol binario que aparece en la figura 12.22 se despliega de la siguiente manera:

```
                  99
             97
                  92
        83
                  72
             71
                  69
   49
                  44
             40
                  32
        28
                  19
             18
                  11
```

Observe que el nodo hoja más a la derecha aparece en la cima de la salida de la columna más a la derecha, y que el nodo raíz aparece a la izquierda de la salida. Cada columna de salida inicia cinco espacios a la derecha de la columna anterior. La función **`despliegaArbol`** debe recibir como argumentos un apuntador al nodo raíz del árbol, y un entero **`espaciosTotales`** que represente el número de espacios que precede al valor a desplegar (esta variable debe comenzar en cero, para que el nodo raíz se despliegue a la izquierda de la pantalla). La función utiliza un recorrido modificado inorden para desplegar el árbol; éste comienza en el nodo más a la derecha del árbol, y trabaja hacia atrás a la izquierda. El algoritmo es el siguiente:

Mientras el apuntador al nodo actual no sea nulo.
Recursivamente llama a **`despliegaArbol`** con el subárbol derecho del nodo actual
 y **`espaciosTotales + 5`**.
Utiliza una instrucción **`for`** para contar de **`1`** hasta **`espaciosTotales`**, y despliega los espacios.
Despliega el valor del nodo actual.
Establece el apuntador al nodo actual para que apunte hacia el subárbol izquierdo del nodo actual.
Incrementa en **`5`** a **`espaciosTotales`**.

## *SECCIÓN ESPECIAL: CÓMO CONSTRUIR SU PROPIO COMPILADOR*

En el ejercicio 7.18, presentamos el Lenguaje Máquina Simpletron (LMS) y creamos el simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección, construimos un compilador que convierte programas escritos en un lenguaje de programación de alto nivel a LMS. Esta sección "une" el proceso completo de programación. Escribiremos programas en este nuevo lenguaje de alto nivel, compilaremos los programas en el compilador, y ejecutaremos los programas en el simulador que construimos en el ejercicio 7.19.

**12.26** (*El lenguaje Simple*.) Antes de que comencemos a construir el compilador, explicaremos un lenguaje de alto nivel sencillo, pero poderoso, parecido a las primeras versiones del popular lenguaje BASIC. A éste le llamamos lenguaje *Simple*. Toda *instrucción* Simple consiste en un *número de línea* y la propia instrucción de Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción comienza con uno de los siguientes comandos Simple: **`rem`**, **`input`**, **`let`**, **`print`**, **`goto`**, **`if...goto`**, o **`end`** (vea la figura 12.23). Todos los comandos, excepto **`end`**, pueden utilizarse repetidamente. Simple evalúa sólo expresiones enteras por medio de los operadores **`+`**, **`-`**, **`*`** y **`/`**. Estos operadores tienen la misma precedencia que en C. Los paréntesis pueden utilizarse para modificar el orden de evaluación de una expresión.

Nuestro compilador Simple reconoce solamente letras minúsculas. Todos los caracteres de un archivo Simple deben estar en minúsculas (las letras mayúsculas ocasionarán un error de sintaxis, a menos que aparezcan en una instrucción **`rem`**, en cuyo caso, se ignoran). Un *nombre de variable* es una sola letra. Simple no permite nombres

| Comando | Instrucción de ejemplo | Descripción |
|---|---|---|
| **`rem`** | **`50 rem este es un comentario`** | El texto que va después de **`rem`** sólo se utiliza con fines de documentación y el compilador lo ignora. |
| **`input`** | **`30 input x`** | Despliega un signo de interrogación para indicar al usuario que introduzca un entero. Lee ese entero desde el teclado, y lo almacena en **`x`**. |
| **`let`** | **`80 let u = 4 * (j – 56)`** | Asigna a **`u`** el valor de **`4 * (j – 56)`**. Observe que una expresión arbitrariamente compleja puede aparecer a la derecha del signo de igual. |
| **`print`** | **`10 print w`** | Despliega el valor de **`w`**. |
| **`goto`** | **`70 goto 45`** | Transfiere el control del programa a la línea **`45`**. |
| **`if...goto`** | **`35 if i == z goto 80`** | Compara si **`i`** y **`z`** son iguales, y transfiere el control del programa a la línea **`80`** si la condición es verdadera; de lo contrario, continúa la ejecución con la siguiente instrucción. |
| **`end`** | **`99 end`** | Termina la ejecución del programa. |

**Figura 12.23** Comandos de Simple.

```
10  rem   determina e imprime la suma de dos enteros
15  rem
20  rem   introduce los dos enteros
30  input a
40  input b
45  rem
50  rem   suma los enteros y almacena el resultado en c
60  let c = a + b
65  rem
70  rem   imprime el resultado
80  print c
90  rem   termina la ejecución del programa
99  end
```

**Figura 12.24** Determina la suma de dos enteros.

de variables descriptivos, por lo que las variables deben explicarse en comentarios para indicar su uso en el programa. Simple sólo utiliza variables enteras, y no tiene declaraciones de variables; el simple hecho de mencionar un nombre de variable en un programa ocasiona que dicha variable se declare e inicialice automáticamente en cero. La sintaxis de Simple no permite manipulación de cadenas (leer, escribir, comparar cadenas, etcétera). Si se encuentra una cadena en un programa de Simple (después de un comando diferente de **rem**), el compilador genera un error de sintaxis. Nuestro compilador asumirá que los programas en Simple se introducen correctamente. El ejercicio 12.29 pide al estudiante que modifique el compilador para que realice una verificación de errores de sintaxis.

Simple utiliza la instrucción condicional **if...goto** y la instrucción no condicional **goto**, para alterar el flujo de control durante la ejecución de un programa. Si la condición de la instrucción **if...goto** es verdadera, el control se transfiere a una línea específica del programa. Los siguientes operadores de relación y de igualdad son válidos en una instrucción **if...goto**: **<**, **>**, **<=**, **>=**, **==** o **!=**. La precedencia de estos operadores es la misma que en C.

Ahora consideremos diversos programas en Simple que muestran las características de Simple. El primer programa (figura 12.24) lee dos enteros desde el teclado, almacena los valores en las variables **a** y **b**, y calcula e imprime su suma (la cual almacena en la variable **c**).

La figura 12.25 determina e imprime el mayor de dos enteros. Los enteros se introducen desde el teclado y se almacenan en **s** y **t**. La instrucción **if...goto** evalúa la condición **s>=t**. Si la condición es verdadera, el control se trasfiere a la línea **90** y **s** se despliega; de lo contrario, **t** se despliega y el control se transfiere a la instrucción **end** de la línea **99**, en donde el programa termina.

Simple no proporciona una estructura de repetición (como las de C, **for**, **while** o **do...while**). Sin embargo, Simple puede simular cada una de las estructuras de repetición de C, utilizando instrucciones **if...goto** y **goto**. La figura 12.26 utiliza un ciclo controlado por centinela para calcular el cuadrado de diversos enteros. Ca-

```
10  rem   determina el mayor de dos enteros
20  input s
30  input t
32  rem
35  rem   evalúa si s >= t
40  if s >= t goto 90
45  rem
50  rem   t es mayor que s, por lo que se imprime t
60  print t
70  goto 99
75  rem
80  rem   s es mayor o igual que t, por lo que se imprime s
90  print s
99  end
```

**Figura 12.25** Encuentra el mayor de dos enteros.

```
10 rem  calcula el cuadrado de diversos enteros
20 input j
23 rem
25 rem  evalúa si se trata del valor centinela
30 if j == -9999 goto 99
33 rem
35 rem  calcula el cuadrado de j y asigna el resultado a k
40 let k = j * j
50 print k
53 rem
55 rem  hace un ciclo para obtener el siguiente j
60 goto 20
99 end
```

**Figura 12.26** Calcula el cuadrado de diversos enteros.

da entero se introduce desde el teclado y se almacena en la variable **`j`**. Si el valor introducido es el centinela **`-9999`**, el control se transfiere a la línea **`99`**, en donde el programa finaliza. De lo contrario, a **`k`** se le asigna el cuadrado de **`j`**, **`k`** se despliega en la pantalla y el control pasa a la línea **`20`**, en donde se introduce el siguiente entero.

Utilizando como guía los programas de ejemplo de las figuras 12.24, 12.25 y 12.26, escriba un programa en Simple para realizar las siguientes tareas:

a) Introduzca tres enteros, determine su promedio e imprima el resultado.
b) Utilice un ciclo controlado por centinela para introducir 10 enteros y calcular e imprimir su suma.
c) Utilice un ciclo controlado por contador para introducir siete enteros, unos positivos y otros negativos, y calcular e imprimir su promedio.
d) Introduzca una serie de enteros y determine e imprima el mayor. El primer entero introducido indica cuántos números deben procesarse.
e) Introduzca 10 enteros e imprima el menor.
f) Calcule e imprima la suma de los enteros pares del 2 al 30.
g) Calcule e imprima el producto de los enteros nones del 1 al 9.

**12.27** (*Construcción de un compilador. Prerrequisito: complete los ejercicios 7.18, 7.19, 12.12, 12.13 y 12.26*.) Ahora que ya presentamos el lenguaje Simple, explicaremos cómo construir nuestro compilador Simple. Primero, considere el proceso por medio del cual un programa en Simple se convierte a LMS y se ejecuta con el simulador Simpletron (vea la figura 12.27). El compilador lee y convierte un archivo que contiene un programa en Simple a código SML. El código LMS se envía a un archivo en disco, en el que las instrucciones LMS aparecen una por línea. Después, el archivo LMS se carga en el simulador Simpletron, y los resultados se envían a un archivo en disco y a la pantalla. Observe que el programa Simpletron desarrollado en el ejercicio 7.19 toma su entrada desde el teclado. Éste debe modificarse para leer desde un archivo, para que pueda ejecutar los programas producidos por nuestro compilador.

El compilador realiza dos *pasadas* al programa en Simple para convertirlo a LMS. La primera pasada construye una tabla de símbolos, en la que cada *número de línea*, *nombre de variable* y *constante* del programa en Simple se almacena con su tipo y su correspondiente ubicación en el código final SML (más adelante explicaremos con detalle la tabla de símbolos). La primera pasada también produce las instrucciones correspondientes en LMS para

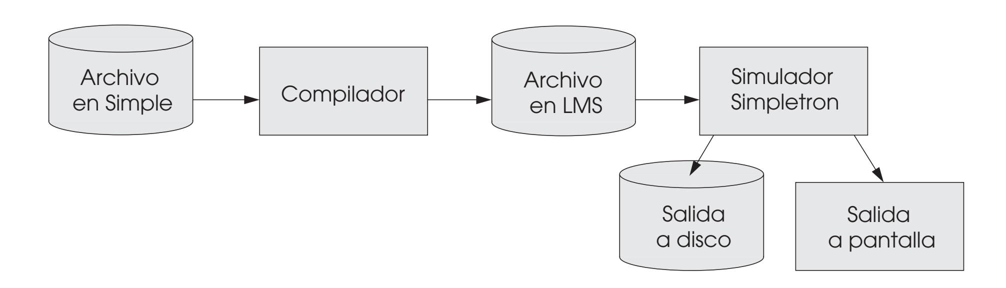



**Figura 12.27** Escritura, compilación y ejecución de un programa en lenguaje Simple.

cada instrucción en Simple. Como veremos, si el programa en Simple contiene instrucciones que transfieren el control a una línea posterior del programa, la primera pasada resulta en un programa LMS que contiene algunas instrucciones incompletas. La segunda pasada del compilador localiza y completa las instrucciones incompletas, y envía el programa LMS a un archivo.

### *Primera pasada*

El compilador comienza leyendo una instrucción del programa en Simple desde memoria. La línea debe separarse en sus tokens individuales (es decir, en "piezas" de una instrucción), para procesarla y compilarla (para facilitar esta tarea, podemos utilizar la función **strtok** de la biblioteca estándar). Recuerde que toda instrucción comienza con un número de línea seguido por un comando. Conforme el compilador separa una instrucción en tokens, si el token es un número de línea, una variable o una constante, ésta se coloca en la tabla de símbolos. Un número de línea sólo se coloca en la tabla de símbolos, si es el primer token de una instrucción. La **tablaSimbolos** es un arreglo de estructuras **entradaTabla** que representa a cada símbolo del programa. No existe restricción alguna con respecto al número de símbolos que puede aparecer en el programa. Por lo tanto, **tablaSimbolos** para un programa en particular podría ser larga. Por ahora, haga que **tablaSimbolos** sea un arreglo de 100 elementos. Usted puede incrementar o reducir su tamaño, una vez que el programa esté funcionando.

La definición de la estructura **entradaTabla** es la siguiente:

```
struct entradaTabla {
    int simbolo;
    char tipo; /* 'C', 'L' o 'V' */
    int ubicacion; /* 00 a 99 */
};
```

Cada estructura **entradaTabla** contiene tres miembros. El miembro **simbolo** es un entero que contiene la representación ASCII de una variable (recuerde que los nombres de variables constan de un solo carácter), de un número de línea o de una constante. El miembro **tipo** es uno de los siguientes caracteres, los cuales indican el tipo del símbolo: **'C'** para una constante, **'L'** para un número de línea, o **'V'** para una variable. El miembro **ubicacion** contiene la ubicación en memoria Simpletron (**00** a **99**) a la que el símbolo hace referencia. La memoria Simpletron es un arreglo de 100 enteros en el que se almacenan las instrucciones y los datos LMS. Para un número de línea, la ubicación es el elemento del arreglo memoria Simpletron en el que comienzan las instrucciones LMS para la instrucción en Simple. Para una variable o constante, la ubicación es el elemento del arreglo memoria Simpletron en el que la variable o constante está almacenada. Las variables y constantes se asignan desde el final de la memoria Simpletron hacia atrás. La primera variable o constante se almacena en la ubicación **99**, la siguiente en **98**, etcétera.

La tabla de símbolos juega un papel importante en la conversión de programas en Simple a LMS. En el capítulo 7 aprendimos que una instrucción LMS es un entero de cuatro dígitos que consta de dos partes: el *código de operación* y el *operando*. El código de operación es definido por comandos en Simple. Por ejemplo, el comando sencillo **input** corresponde al código de operación LMS **10** (lee), el comando **print** corresponde al código **11** (escribe). El operando es una ubicación en memoria que contiene los datos sobre los que el código de operación realiza su tarea (por ejemplo, el código de operación **10** lee un valor desde el teclado y lo almacena en la ubicación de memoria especificada por el operando). El compilador busca **tablaSimbolos** para determinar la ubicación de memoria Simpletron para cada símbolo, de tal forma que la ubicación correspondiente pueda utilizarse para completar las instrucciones de LMS.

La compilación de cada instrucción LMS se basa en su comando. Por ejemplo, después de que el número de línea correspondiente a una instrucción **rem** se inserta en la tabla de símbolos, el compilador ignora el resto de la instrucción, ya que un comentario sólo sirve para documentación. Las instrucciones **input**, **print**, **goto** y **end**, corresponden a las instrucciones de LMS *read*, *write*, *branch* (hacia una ubicación específica) y *halt*. Las instrucciones que contienen estos comandos de Simple se convierten directamente a LMS. [*Nota:* Una instrucción **goto** puede contener una referencia no resuelta, si el número de línea especificado hace referencia a una instrucción más avanzada dentro del archivo correspondiente al programa en Simple; en ocasiones, a esto se le llama referencia adelantada.]

Cuando se compila una instrucción **goto** con una referencia no resuelta, a la instrucción LMS se le debe *colocar una bandera* para indicar que la segunda pasada del compilador debe completar la instrucción. Las banderas se almacenan en el arreglo de tipo **entero** de 100 elementos llamado **banderas**, en el que cada elemento se inicializa en **–1**. Si la ubicación en memoria a la que hace referencia un número de línea del programa en Simple aún no se conoce (es decir, no se encuentra en la tabla de símbolos), el número de línea se almacena en el arreglo **banderas** en el elemento que tiene el mismo subíndice que la instrucción incompleta. El operando de la instrucción incompleta se establece temporalmente en **00**. Por ejemplo, una instrucción no condicional bifurcar (que hace una

referencia adelantada) se deja como **+4000**, hasta la segunda pasada del compilador. En un momento describiremos la segunda pasada del compilador.

La compilación de instrucciones **if...goto** y **let** es más complicada que la de otras instrucciones; éstas son las únicas instrucciones que producen más de una instrucción LMS. Por una instrucción **if...goto**, el compilador produce código para evaluar la condición y para ramificarse hacia otra línea, en caso necesario. El resultado de la ramificación podría ser una referencia no resuelta. Cada uno de los operadores de relación y de igualdad puede simularse por medio de las instrucciones de LMS *branch zero* y *branch negative* (o posiblemente una combinación de ambas).

Para una instrucción **let**, el compilador produce código para evaluar una expresión aritmética arbitrariamente compleja que conste de variables enteras y/o constantes. Las expresiones deben separar cada operando y operador con espacios. Los ejercicios 12.12 y 12.13 presentaron el algoritmo de conversión de infijo a postfijo y el de evaluación de postfijos que utilizan los compiladores para evaluar expresiones. Antes de continuar con su compilador, debe completar cada uno de estos ejercicios. Cuando un compilador encuentra una expresión, éste la convierte de notación infijo a postfijo, y después evalúa la expresión en postfijo.

¿Cómo es que el compilador produce el lenguaje máquina para evaluar una expresión que contiene variables? El algoritmo de evaluación postfijo contiene un "gancho" que permite a nuestro compilador generar instrucciones LMS, en lugar de realmente evaluar la expresión. Para aceptar a este "gancho" en el compilador, el algoritmo de evaluación postfijo debe modificarse para que busque en la tabla de símbolos cada símbolo que encuentre (y que posiblemente lo inserte), que determine la ubicación en memoria correspondiente a ese símbolo, y que *meta la ubicación de memoria en la pila, en lugar del símbolo*. Cuando se encuentra un operador en la expresión postfijo, las dos ubicaciones de memoria en la cima de la pila son eliminadas, y se produce lenguaje máquina para que efectúe la operación, utilizando como operando las ubicaciones de memoria. El resultado de cada subexpresión se almacena en una ubicación de memoria temporal y se mete nuevamente en la pila para que la evaluación de la expresión postfijo pueda continuar. Cuando se completa la evaluación postfijo, la posición de memoria que contiene el resultado es la única ubicación que se deja en la pila. Ésta se saca, y se generan instrucciones LMS para asignar el resultado a la variable que se encuentra a la izquierda de la instrucción **let**.

### *Segunda pasada*

La segunda pasada del compilador realiza dos tareas: resuelve cualquier referencia no resuelta y envía el código LMS a un archivo. La resolución de referencias ocurre de la siguiente manera:

1) Busca en el arreglo **banderas** alguna referencia no resuelta (es decir, un elemento con un valor diferente de **-1**).
2) Localiza en el arreglo **tablaSimbolos** la estructura que contenga el símbolo almacenado en el arreglo **banderas** (asegúrese de que el tipo del símbolo sea **'L'**, en el caso de un número de línea).
3) Inserte la ubicación de memoria, desde el miembro **ubicacion**, en la instrucción que contiene la referencia no resuelta (recuerde que una instrucción que contiene una referencia no resuelta tiene el operando **00**).
4) Repita los pasos 1, 2 y 3, hasta que se alcance el final del arreglo **banderas**.

Después de que se completa el proceso de resolución, el arreglo completo que contiene el código LMS se envía a un archivo en disco con una instrucción LMS por línea. Este archivo puede leerse para su ejecución con el Simpletron (después de que el simulador se modifique para que lea su entrada desde un archivo).

### *Un ejemplo completo*

El siguiente ejemplo ilustra una conversión completa de un programa en Simple a LMS, tal como la realizaría el compilador de Simple. Considere un programa en Simple que introduce un entero y suma los valores entre 1 y ese entero. El programa y las instrucciones LMS producidas por la primera pasada aparecen en la figura 12.28. La tabla de símbolos construida por la primera pasada, aparece en la figura 12.29.

| Programa en Simple | Ubicación e instrucción SML | Descripción |
|---|---|---|
| **5 rem suma 1 a x** | *ninguna* | **rem** ignorado |
| **10 input x** | **00 +1099** | lee **x** y lo coloca en la posición **99** |

**Figura 12.28** Instrucciones SML producidas después de la primera pasada del compilador. (Parte 1 de 2.)

| Programa en Simple | Ubicación e instrucción SML | Descripción |
|---|---|---|
| **15 rem verifica que y == x** | *ninguna* | **rem** ignorado |
| **20 if y == x goto 60** | **01 +2098** | carga **y(98)** en un acumulador |
| | **02 +3199** | resta **x(99)** del acumulador |
| | **03 +4200** | *si el resultado es cero, ramifica hacia una ubicación no resuelta* |
| **25 rem incrementa y** | *ninguna* | **rem** ignorado |
| **30 let y = y + 1** | **04 +2098** | carga **y** en un acumulador |
| | **05 +3097** | suma **1(97)** al acumulador |
| | **06 +2196** | almacena **96** en una ubicación temporal |
| | **07 +2096** | carga **96** desde la ubicación temporal |
| | **08 +2198** | almacena en **y** al acumulador |
| **35 rem suma y al total** | *ninguna* | **rem** ignorado |
| **40 let t = t + y** | **09 +2095** | carga **t(95)** en el acumulador |
| | **10 +3098** | suma **y** al acumulador |
| | **11 +2194** | almacena **94** en una ubicación temporal |
| | **2 +2094** | carga **94** desde la ubicación temporal |
| | **13 +2195** | almacena el acumulador en **t** |
| **45 rem ciclo sobre y** | *ninguna* | **rem** ignorado |
| **50 goto 20** | **14 +4001** | ramifica hacia la ubicación **01** |
| **55 rem despliega resultado** | *ninguna* | **rem** ignorado |
| **60 print t** | **15 +1195** | despliega **t** en la pantalla |
| **99 end** | **16 +4300** | termina la ejecución |

**Figura 12.28** Instrucciones LMS producidas después de la primera pasada del compilador. (Parte 2 de 2.)

| Símbolo | Tipo | Ubicación |
|---|---|---|
| **5** | **L** | **00** |
| **10** | **L** | **00** |
| **'x'** | **V** | **99** |
| **15** | **L** | **01** |
| **20** | **L** | **01** |
| **'y'** | **V** | **98** |
| **25** | **L** | **04** |
| **30** | **L** | **04** |

**Figura 12.29** Tabla de símbolos para el programa de la figura 12.28. (Parte 1 de 2.)

| Símbolo | Tipo | Ubicación |
|---|---|---|
| `1` | `C` | `97` |
| `35` | `L` | `09` |
| `40` | `L` | `09` |
| `'t'` | `V` | `95` |
| `45` | `L` | `14` |
| `50` | `L` | `14` |
| `55` | `L` | `15` |
| `60` | `L` | `15` |
| `99` | `L` | `16` |

**Figura 12.29** Tabla de símbolos para el programa de la figura 12.28. (Parte 2 de 2.)

La mayoría de las instrucciones en Simple se convierten directamente en instrucciones sencillas de LMS. Las excepciones en este programa son los comentarios, la instrucción **`if...goto`** de la línea **`20`** y las instrucciones **`let`**. Los comentarios no se traducen en lenguaje máquina. Sin embargo, el número de línea de un comentario se coloca en la tabla de símbolos, en caso de que se haga referencia a dicho número de línea en una instrucción **`goto`** o en una **`if...goto`**. La línea **`20`** del programa especifica que si la condición **`y == x`** es verdadera, el control del programa se transfiere a la línea **`60`**. Debido a que la línea **`60`** aparece más adelante en el programa, la primera pasada del compilador todavía no ha colocado **`60`** en la tabla de símbolos (los números de línea se colocan en la tabla de símbolos solamente cuando aparecen como el primer token de una instrucción). Por lo tanto, no es posible en este momento determinar el operando de la instrucción de LMS *branch zero* en la ubicación **`03`** del arreglo de instrucciones LMS. El compilador coloca **`60`** en la ubicación **`03`** del arreglo **`banderas`** para indicar que la segunda pasada completará esta instrucción.

Debemos dar seguimiento a la siguiente ubicación de la instrucción en el arreglo LMS, ya que no hay una correspondencia uno a uno entre instrucciones Simple e instrucciones LMS. Por ejemplo, la instrucción **`if...goto`** de la línea **`20`** se compila en tres instrucciones LMS. Cada vez que se produce una instrucción, debemos incrementar el *contador de instrucciones* hacia la siguiente ubicación en el arreglo LMS. Observe que el tamaño de la memoria del Simpletron podría representar un problema para programas en Simple con demasiadas instrucciones, variables y constantes. Es probable que el compilador se quede sin memoria. Para evaluar este caso, su programa debe tener un *contador de datos* que dé seguimiento a la ubicación del arreglo LMS en la que la siguiente variable o constante se almacenará. Si el valor de la instrucción contador es mayor que el valor del contador de datos, el arreglo LMS está lleno. En este caso, el proceso de compilación debe terminar y el compilador debe imprimir un mensaje de error que indique que se quedó sin memoria durante la compilación.

### *Visión paso a paso del proceso de compilación*

Ahora veamos el proceso de compilación del programa en Simple de la figura 12.28. El compilador lee la primera línea del programa

```
5 rem suma 1 a x
```

desde memoria. El primer token de la instrucción (el número de línea) se determina por medio de **`strtok`** (vea el capítulo 8 para una explicación de las funciones para manipulación de cadenas en C). El token devuelto por **`strtok`** se convierte en un entero utilizando **`atoi`**, por lo que el símbolo **`5`** puede localizarse en la tabla de símbolos. Si el símbolo no se encuentra, éste se inserta en la tabla de símbolos. Debido a que nos encontramos al principio del programa y a que ésta es la primera línea, aún no hay símbolos en la tabla. Entonces, **`5`** se inserta en la tabla de símbolos como de tipo **`L`** (número de línea), y se asigna a la primera ubicación del arreglo LMS (**`00`**). Aunque esta línea es un comentario, por el número de línea se asigna un espacio en la tabla de símbolos (en caso de que se haga referencia a él en una instrucción **`goto`** o en una **`if...goto`**). Una instrucción **`rem`** no genera instrucción LMS alguna, por lo que el contador de instrucciones no se incrementa.

Después, la instrucción

```
10 input x
```

se separa en **tokens**. El número de línea **10** se coloca en la tabla de símbolos como de tipo **L**, y se asigna en la primera ubicación del arreglo LMS (**00**, ya que un comentario inició el programa, y el contador de instrucciones es actualmente **00**). El comando **input** indica que el siguiente token es una variable (sólo una variable puede aparecer en una instrucción **input**). Debido a que **input** corresponde directamente a una operación en código LMS, el compilador simplemente tiene que determinar la ubicación de **x** en el arreglo LMS. El símbolo **x** no se encontró en la tabla de símbolos, por lo que se inserta en dicha tabla como la representación ASCII de **x**, se le da el tipo **V**, y se le asigna la ubicación **99** del arreglo LMS (el almacenamiento de datos comienza en **99** y se asigna hacia atrás). Ahora, esta instrucción puede generar código LMS. El código de operación **10** (el código de operación de lectura de LMS) se multiplica por **100**, y la ubicación de **x** (como se determinó en la tabla de símbolos) se suma para completar la instrucción. Después, la instrucción se almacena en la ubicación **00** del arreglo LMS. El contador de instrucciones se incrementa en **1**, ya que se produjo una instrucción LMS.

Después, la instrucción

```
15 rem    verifica y == x
```

se separa en tokens. Se busca en la tabla de símbolos el número de línea **15** (el cual no se encuentra). El número de línea se inserta como de tipo **L**, y se asigna a la siguiente ubicación del arreglo, **01** (recuerde que las instrucciones **rem** no producen código, por lo que el contador de instrucciones no se incrementa).

Después se separa en tokens la instrucción

```
20 if y == x goto 60
```

El número de línea **20** se inserta en la tabla de símbolos y se le da el tipo **L**, con la siguiente posición en el arreglo LMS, **01**. El comando **if** indica que se va a evaluar una condición. La variable **y** no se encuentra en la tabla de símbolos, por lo que se inserta en ella y se le da el tipo **V** y la ubicación **98**. Posteriormente, se generan instrucciones SML para evaluar la condición. Debido a que no hay un equivalente directo en SML para **if...goto**, ésta debe simularse realizando un cálculo que utilice **x** y **y**, y que realice una ramificación basada en el resultado. Si **y** es igual que **x**, el resultado de restar **x** de **y** es cero, por lo que la instrucción *branch zero* puede utilizarse con el resultado del cálculo para simular la instrucción **if...goto**. El primer paso requiere que **y** se cargue (desde la ubicación **98** de SML) en el acumulador. Esto produce la instrucción **01 +2098**. Después, **x** se resta del acumulador. Esto produce la instrucción **02 +3199**. El valor del acumulador puede ser cero, positivo o negativo. Debido a que el operador es **==**, queremos utilizar *branch zero*. Primero, se busca en la tabla de símbolos la ubicación ramificada (en este caso **60**), la cual no se encuentra. Entonces, **60** se coloca en el arreglo **banderas** en la ubicación **03**, y se genera la instrucción **03 +4200** (no podemos sumar la ubicación ramificada debido a que aún no hemos asignado una ubicación a la línea **60** en el arreglo SML). El contador de instrucciones se incrementa a **04**.

El compilador continúa con la instrucción

```
25 rem incrementa y
```

El número de línea 25 se inserta en la tabla de símbolos como de tipo **L** y se le asigna la ubicación **04** en SML. El contador de instrucciones no se incrementa.

Cuando la instrucción

```
30 let y = y + 1
```

se separa en tokens, el número de línea **30** se inserta en la tabla de símbolos como de tipo **L** y se le asigna la ubicación **04**. El comando **let** indica que la línea es una instrucción de asignación. Primero, todos los símbolos de la línea se insertan en la tabla de símbolos (si aún no están ahí). El entero **1** se agrega a la tabla de símbolos como de tipo **C** y se le asigna la ubicación **97**. Después, el lado derecho de la asignación se convierte de notación infijo a notación postfijo. Luego, se evalúa la expresión postfijo **(y 1 +)**. El símbolo **y** se localiza en la tabla de símbolos, y su ubicación en memoria se mete en la pila. El símbolo **1** también se localiza en la tabla de símbolos, y su ubicación en memoria se mete en la pila. Cuando se encuentra el operador **+**, el evaluador postfijo saca la pila hacia el operando derecho del operador, y saca nuevamente la pila hacia el operando izquierdo del operador, después produce las instrucciones SML

**04 +2098** (*carga y*)
**05 +3097** (*suma 1*)

El resultado de la expresión se almacena en una ubicación temporal de memoria (**96**) con la instrucción

**06 +2196** (*almacena temporalmente*)

y la ubicación temporal se mete en la pila. Ahora que la expresión se evaluó, el resultado debe almacenarse en **y** (es decir, en la variable del lado izquierdo del **=**). Entonces, la ubicación temporal se carga en el acumulador y éste se almacena en **y** con las instrucciones

**07 +2096** (*carga temporalmente*)
**08 +2198** (*almacena y*)

El lector notará inmediatamente que las instrucciones SML parecen redundantes. En un momento explicaremos este asunto.

Cuando la instrucción

```
35 rem suma y al total
```

se separa en tokens, el número de línea **35** se inserta en la tabla de símbolos como de tipo **L** y se le asigna la posición **09**.

La instrucción

```
40 let t = t + y
```

es parecida a la línea **30**. La variable **t** se inserta en la tabla de símbolos como de tipo **V** y se le asigna la ubicación **95**. Las instrucciones siguen la misma lógica y formato que la línea **30**, y se generan las instrucciones **09 +2095**, **10 +3098**, **11 +2194**, **12 +2094** y **13 +2195**. Observe que el resultado de **t + y** se asigna a la ubicación temporal **94** antes de que se asigne a **t(95)**. Una vez más, el lector notará que las instrucciones que se encuentran en las ubicaciones de memoria **11** y **12** parecen redundantes. De nuevo, esto lo explicaremos en un momento.

La instrucción

```
45 rem ciclo sobre y
```

es un comentario, por lo que la línea **45** se agrega a la tabla de símbolos como de tipo **L** y se le asigna la ubicación **14**.

La instrucción

```
50 goto 20
```

transfiere el control a la línea **20**. El número de línea **50** se inserta en la tabla de símbolos como de tipo **L** y se le asigna la ubicación **SML 14**. La instrucción equivalente de **goto** en SML es la *instrucción no condicional branch* (**40**), la cual transfiere el control a una ubicación SML específica. El compilador busca en la tabla de símbolos a la línea **20** y encuentra que ésta corresponde a la ubicación **SML 01**. El código de operación **(40)** se multiplica por **100** y la ubicación **01** se agrega a él para producir la instrucción **14 +4001**.

La instrucción

```
55 rem despliega resultado
```

es un comentario, por lo que la línea **55** se inserta en la tabla de símbolos como de tipo **L** y se le asigna la ubicación **SML 15**.

La instrucción

```
60 print t
```

es una instrucción de salida. El número de línea **60** se inserta en la tabla de símbolo como de tipo **L** y se le asigna la ubicación **15**. El equivalente de **print** en SML es el código de operación **11** (*escribir*). La ubicación de **t** se determina a partir de la tabla de símbolos y se agrega al resultado del código de operación multiplicado por **100**.

La instrucción

```
99 end
```

es la línea final del programa. El número de línea **99** se almacena en la tabla de símbolos como de tipo **L** y se le asigna la ubicación **SML 16**. El comando **end** produce la instrucción **SML +4300** (**43** es *halt* en SML), la cual se escribe como la instrucción final en el arreglo memoria **SML**.

Esto completa la primera pasada del compilador. Ahora consideraremos la segunda pasada. Se busca en el arreglo **banderas** cualquier valor diferente de **-1**. La ubicación **03** contiene **60**, por lo que el compilador sabe que la instrucción **03** está incompleta. El compilador completa la instrucción buscando **60** en la tabla de símbolos, determina su ubicación y la agrega a la instrucción incompleta. En este caso, la búsqueda determina que la línea **60** corresponde a la ubicación **15**, por lo que la instrucción completa **03 +4215** se produce y reemplaza a **03 +4200**. Ahora, el programa en Simple se compiló con éxito.

Para construir el compilador, tendrá que realizar cada una de las siguientes tareas:

a) Modifique el programa simulador Simpletron que escribió en el ejercicio 7.19 para que tome su entrada desde un archivo especificado por el usuario (vea el capítulo 11). Además, el simulador debe enviar sus resultados a un archivo en disco en el mismo formato que el desplegado en pantalla.

b) Modifique el algoritmo de evaluación infijo a postfijo del ejercicio 12.12 para procesar operandos enteros de varios dígitos y operandos de nombres de variables de una sola letra. [*Pista:* Puede utilizar la función **`strtok`** de la biblioteca estándar para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros por medio de la función **`atoi`** de la biblioteca estándar.] [*Nota:* La representación de datos de la expresión postfijo debe modificarse para que soporte nombres de variables y constantes enteras.]

c) Modifique el algoritmo de evaluación postfijo para procesar operandos enteros de varios dígitos y operandos de nombres de variables. Además, el algoritmo debe ahora implementar el "gancho" que explicamos anteriormente, para que las instrucciones SML se produzcan, en lugar de evaluar directamente la expresión. [*Pista:* Puede utilizar la función **`strtok`** de la biblioteca estándar para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros por medio de la función **`atoi`** de la biblioteca estándar.] [*Nota:* La representación de datos de la expresión postfijo debe modificarse para que soporte nombres de variables y constantes enteras.]

d) Construya el compilador. Incorpore las partes (b) y (c) para evaluar expresiones de instrucciones **`let`**. Su programa debe contener una función que realice la primera pasada del compilador, y una función que realice la segunda pasada. Ambas funciones pueden llamar otras funciones para llevar a cabo sus tareas.

**12.28** (*Optimización del compilador Simple*.) Cuando un programa se compila y se convierte en LMS, se genera un conjunto de instrucciones. Ciertas combinaciones de instrucciones con frecuencia se repiten, por lo general en tercias conocidas como *producciones*. Una producción normalmente consiste en tres instrucciones como *load, add* y *store*. Por ejemplo, la figura 12.30 ilustra cinco de las instrucciones LMS que se produjeron en la compilación del programa de la figura 12.28. Las tres primeras instrucciones forman la producción que suma **`1`** a **`y`**. Observe que las instrucciones **`06`** y **`07`** almacenan el valor del acumulador en la ubicación temporal **`96`**, y después cargan de vuelta el valor en el acumulador, de tal forma que la instrucción **`08`** pueda almacenar el valor en la ubicación **`98`**. Con frecuencia, una producción va seguida de una instrucción load para la misma ubicación en la que fue almacenada. Este código puede *optimizarse* eliminando la instrucción store y la subsiguiente instrucción **`load`** que operan en la misma ubicación de memoria. Esta optimización permitiría al Simpletron ejecutar el programa más rápidamente, ya que hay menos instrucciones en esta versión. La figura 12.31 muestra la optimización del SML para el programa de la figura 12.28. Observe que en el código optimizado hay cuatro instrucciones menos; un ahorro de memoria del 25%.

Modifique el compilador para proporcionar una opción para optimizar el código en Lenguaje Máquina Simpletron que éste produce. Manualmente compare el código no optimizado con el optimizado, y calcule el porcentaje de reducción.

```
04 +2098     (load)
05 +3097     (add)
06 +2196     (store)
07 +2096     (load)
08 +2198     (store)
```

**Figura 12.30** Código no optimizado del programa correspondiente a la figura 12.28.

| Programa en Simple | Ubicación e instrucción SML | Descripción |
|---|---|---|
| **`5 rem suma 1 a x`** | *ninguna* | **`rem`** ignorado |
| **`10 input x`** | **`00 +1099`** | lee **`x`** y lo coloca en la posición **`99`** |
| **`15 rem verifica que y == x`** | *ninguna* | **`rem`** ignorado |
| **`20 if y == x goto 60`** | **`01 +2098`** | carga **`y(98)`** en un acumulador |

**Figura 12.31** Código optimizado para el programa de la figura 12.28. (Parte 1 de 2.)

| Programa en Simple | Ubicación e instrucción SML | Descripción |
|---|---|---|
| | **02 +3199** | resta **x(99)** del acumulador |
| | **03 +4211** | si el resultado es cero, ramifica hacia la ubicación **11** |
| **25 rem incrementa y** | *ninguna* | **rem** ignorado |
| **30 let y = y + 1** | **04 +2098** | carga **y** en un acumulador |
| | **05 +3097** | suma **1(97)** al acumulador |
| | **06 +2198** | almacena el acumulador en **y(98)** |
| **35 rem suma y al total** | *ninguna* | **rem** ignorado |
| **40 let t = t + y** | **07 +2096** | carga **t** desde la ubicación **(96)** |
| | **08 +3098** | suma **y(98)** al acumulador |
| | **09 +2196** | almacena el acumulador en **t(96)** |
| **45 rem ciclo sobre y** | *ninguna* | **rem** ignorado |
| **50 goto 20** | **10 +4001** | ramifica hacia la ubicación **01** |
| **55 rem despliega resultado** | *ninguna* | **rem** ignorado |
| **60 print t** | **11 +1196** | despliega **t(96)** en la pantalla |
| **99 end** | **12 +4300** | termina la ejecución |

**Figura 12.31** Código optimizado para el programa de la figura 12.28. (Parte 2 de 2.)

**12.29** (*Modificaciones al compilador Simple*.) Realice las siguientes modificaciones al compilador Simple. Algunas de estas modificaciones pueden requerir también algunas modificaciones al programa del simulador Simpletron escrito en el ejercicio 7.19.

a) Permita que el operador módulo (**%**) se utilice en las instrucciones **let**. El Lenguaje Máquina Simpletron debe modificarse para incluir una instrucción módulo.
b) Permita la exponenciación en una instrucción **let**, por medio del operador de exponenciación **^**. El Lenguaje Máquina Simpletron debe modificarse para incluir una instrucción de exponenciación.
c) Permita que el compilador reconozca letras mayúsculas y minúsculas en instrucciones Simple (por ejemplo, **'A'** es equivalente a **'a'**). No se necesitan modificaciones al simulador de Simpletron.
d) Permita que las instrucciones **input** lean valores para múltiples variables, como **input x**, **y**. No se necesitan modificaciones al simulador de Simpletron.
e) Permita que el compilador despliegue múltiples valores en una sola instrucción **print**, como **print a**, **b**, **c**. No se necesitan modificaciones al simulador de Simpletron.
f) Agregue capacidades de verificación de sintaxis al compilador, para que se desplieguen mensajes de error cuando se encuentren errores de sintaxis en un programa en Simple. No se necesitan modificaciones al simulador de Simpletron.
g) Permita arreglos de enteros. No se necesitan modificaciones al simulador de Simpletron.
h) Permita subrutinas especificadas por los comandos de Simple, **gosub** y **return**. El comando **gosub** pasa el control del programa a una subrutina, y el comando **return** pasa el control de regreso a la instrucción posterior a la **gosub**. Esto es similar a una llamada de función en C. La misma subrutina puede ser llamada desde muchas **gosubs** distribuidas a lo largo de un programa. No se necesitan modificaciones al simulador de Simpletron.
i) Permita estructuras de repetición de la forma

```
for x = 2 to 10 step 2
    rem instrucciones Simple
    next
```

j) Esta instrucción **for** realiza un ciclo desde 2 hasta 10 con un incremento de 2. La línea **next** marca el final del cuerpo de la línea **for**. No se necesitan modificaciones al simulador de Simpletron.

k) Permita estructuras de repetición de la forma

```
for x = 2 to 10
    rem instrucciones Simple
    next
```

l) Esta instrucción **for** realiza un ciclo desde 2 hasta 10 con un incremento predeterminado de 1. No se necesitan modificaciones al simulador de Simpletron.

m) Permita al compilador procesar la entrada y salida de cadenas. Esto requiere que se modifique al simulador de Simpletron para que procese y almacene valores de cadena. [*Pista:* Cada palabra en Simpletron puede dividirse en dos grupos, cada uno con un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal en ASCII de un carácter.] Agregue una instrucción en lenguaje máquina que imprima una cadena que comience en una cierta ubicación de memoria Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada mitad siguiente de una palabra contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina verifica la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente.

n) Permita al compilador procesar valores de punto flotante además de valores enteros. El simulador de Simpletron también debe modificarse para procesar valores de punto flotante.

**12.30** (*Un intérprete de Simple*.) Un intérprete es un programa que lee una instrucción de un programa en lenguaje de alto nivel, determina la operación a realizar por la instrucción, y la ejecuta de inmediato. El programa no se convierte primero a lenguaje máquina. Los intérpretes ejecutan lentamente, ya que cada instrucción encontrada en el programa primero debe descifrarse. Si las instrucciones se encuentran en un ciclo, éstas se descifran cada vez que son encontradas en el ciclo. Las primeras versiones del lenguaje de programación BASIC se implementaron como intérpretes.

Escriba un intérprete para el lenguaje Simple que explicamos en el ejercicio 12.26. El programa debe utilizar el convertidor de infijo a postfijo que desarrollamos en el ejercicio 12.12 y el evaluador postfijo que desarrollamos en el ejercicio 12.13, para evaluar expresiones en una instrucción **let**. Las mismas restricciones aplicadas en el lenguaje Simple del ejercicio 12.26 deben mantenerse en este programa. Evalúe el intérprete con los programas en Simple escritos en el ejercicio 12.26. Compare los resultados de ejecutar estos programas en el intérprete, con los resultados de compilar los programas en Simple y de ejecutarlos en el simulador de Simpletron construido en el ejercicio 7.19.

# 13

# El preprocesador de C

## Objetivos

- Utilizar **`#include`** para desarrollar programas grandes.
- Utilizar **`#define`** para crear macros con y sin argumentos.
- Comprender la compilación condicional.
- Desplegar mensajes de error durante la compilación condicional.
- Utilizar afirmaciones para evaluar si los valores de las expresiones son correctos.




*Mantén el bien, defínelo bien.*
Alfred, Lord Tennyson

*Te encontré un argumento. Pero no estoy obligado a hacerte entender.*
Samuel Johnson

*Un buen símbolo es el mejor argumento, y tiene la misión de persuadir a miles.*
Ralph Waldo Emerson

*Las condiciones son fundamentalmente sonido.*
Herbert Hoover [Diciembre de 1929]

*Al partisano, cuando está comprometido en una disputa, no le importan nada los derechos en cuestión, sólo le importa convencer a sus escuchas de sus propias afirmaciones.*
Platón

## Plan general



## 13.1 Introducción

Este capítulo describe el *preprocesador* de C. El preprocesamiento ocurre antes de la compilación de un programa. Algunas de las acciones que puede realizar son la inclusión de otros archivos dentro del archivo a compilar, la definición de *constantes simbólicas* y *macros*, la *compilación condicional* del código de un programa y la *ejecución condicional de las directivas del preprocesador*. Todas las directivas del preprocesador comienzan con # y, en la misma línea, antes de una directiva solamente pueden aparecer espacios en blanco.

## 13.2 La directiva de preprocesador #include

A lo largo del libro, hemos utilizado la *directiva de preprocesador* **#include**. Esta directiva provoca la inclusión de una copia del archivo especificado en lugar de la directiva. Las dos formas de la directiva **#include** son:

```
#include <nombre de archivo>
#include “nombre de archivo”
```

La diferencia entre ambas es la ubicación en la que el preprocesador busca el archivo a incluir. Si el nombre del archivo se encierra entre comillas, el preprocesador busca el archivo a incluir en el mismo directorio en donde se encuentra el archivo que va a compilarse. Por lo general, este método se utiliza para incluir los encabezados definidos por el programador. Si el nombre del archivo se encierra entre llaves angulares (**<** y **>**), utilizadas por los *encabezados de la biblioteca estándar*, la búsqueda se realiza de acuerdo con la implementación de C, por lo general a través de directorios preestablecidos.

La directiva **#include** se utiliza para incluir encabezados de la biblioteca estándar, tales como **stdio.h** y **stdlib.h** (vea la figura 5.6). Además, la directiva **#include** se utiliza en programas que consisten en varios archivos fuente que van a compilarse juntos. En el archivo, a menudo se crea y se incluye un encabezado que contiene declaraciones comunes para los diferentes archivos del programa. Ejemplos de tales declaraciones son las declaraciones de estructuras y uniones, enumeraciones y prototipos de funciones.

## 13.3 La directiva de preprocesador #define: Constantes simbólicas

La *directiva* **#define** crea *constantes simbólicas* (constantes representadas por símbolos) y *macros* (operaciones definidas como símbolos). El formato de la directiva **#define** es

**#define** *identificador texto de reemplazo*

Cuando esta línea aparece en un archivo, todas las ocurrencias subsecuentes del *identificador*, se reemplazarán automáticamente con el *texto de reemplazo* antes de la compilación del programa. Por ejemplo:

```
#define PI 3.14159
```

reemplaza todas las ocurrencias subsiguientes de la constante simbólica **PI** con la constante numérica **3.14159**. Las constantes simbólicas permiten al programador crear el nombre de una constante y utilizarlo a través del programa. Si la constante necesita modificarse a través del programa, es posible modificarla una vez en la directiva **#define**. Cuando se recompila el programa, todas las ocurrencias de la constante en el programa se modificarán. [*Nota:* Todo lo que se encuentra a la derecha del nombre de la constante simbólica reemplaza a la constante simbólica.] Por ejemplo, **#define PI = 3.14159** provoca que el preprocesador reemplace cada ocurrencia del identificador **PI** con **= 3.14159**. Esto provoca muchos errores de lógica y errores de sintaxis. La redefinición de una constante simbólica con un nuevo valor también es un error.



**Buena práctica de programación 13.1**

*Utilizar nombres significativos para las constantes simbólicas ayuda a hacer programas más autodocumentados.*

## 13.4 La directiva de preprocesador #define: Macros

Una *macro* es un identificador definido dentro de una directiva de preprocesador **#define**. Como en las constantes simbólicas, el *identificador de la macro* se reemplaza en el programa con el *texto de reemplazo* antes de que se compile el programa. Las macros se pueden definir con o sin *argumentos*. Una macro sin argumentos se procesa como una constante simbólica. En una *macro con argumentos*, los argumentos se sustituyen dentro del texto de reemplazo, y después se *desarrolla* la macro; es decir, el texto de reemplazo sustituye al identificador y a la lista de argumentos del programa.

Considere la siguiente definición de una macro con un argumento para el área de un círculo:

```
#define AREA_CIRCULO( x ) ( ( PI ) * ( x ) * ( x ) )
```

Siempre que aparezca **AREA_CIRCULO(y)** en el archivo, el valor de **y** se sustituirá por **x** dentro del texto de reemplazo, la constante simbólica **PI** se reemplaza con su valor (definido previamente) y la macro se desarrolla en el programa. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( 4 );
```

se desarrolla como

```
area = ( ( 3.14159 ) * ( 4 ) * ( 4 ) );
```

y el valor de la expresión se evalúa y se asigna a la variable **area**. Los paréntesis alrededor de cada **x** dentro del texto de reemplazo fuerzan el orden apropiado de evaluación, cuando el argumento de la macro es una expresión. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( c + 2 );
```

se desarrolla como

```
area = ( (3.14159 ) * ( c + 2 ) * ( c + 2 ) );
```

la cual se evalúa correctamente debido a que los paréntesis fuerzan el orden apropiado de evaluación. Si se omiten los paréntesis, el desarrollo de la macro es

```
area = 3.14159 * c + 2 * c + 2;
```

la cual se evalúa incorrectamente como

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

debido a las reglas de precedencia de los operadores.

**Error común de programación 13.1**

*Olvidar encerrar los argumentos de una macro entre paréntesis en el texto de reemplazo, puede provocar errores de lógica.*

La macro **AREA_CIRCULO** podría definirse como una función. La función **areaCirculo**

```
double areaCirculo ( double x )
{
   return  3.14159 * x * x;
}
```

realiza el mismo cálculo que la macro **AREA_CIRCULO**, pero la sobrecarga de una llamada a la función se asocia con **areaCirculo**. Las ventajas de la macro **AREA_CIRCULO** son que las macros insertan el código directamente en el programa (lo que evita la sobrecarga de llamadas a la función) y que el programa se mantiene legible, debido a que el cálculo de **AREA_CIRCULO** se define por separado y se le asigna un nombre significativo. Una desventaja es que su argumento se evalúa dos veces.

**Tip de rendimiento 13.1**

*Algunas veces, las macros pueden utilizarse para reemplazar una llamada a una función con código* ***inline*** *antes del tiempo de ejecución. Esto elimina la sobrecarga de llamadas a la función.*

La siguiente es la definición de una macro con dos argumentos para el área de un rectángulo:

```
#define AREA_RECTANGULO( x, y ) ( ( x ) * ( y ) )
```

Dondequiera que aparezca **AREA_RECTANGULO( x, y )** en el programa, los valores de **x** y **y** se sustituyen en el texto de reemplazo de la macro, y la macro se desarrolla en lugar del nombre de la macro. Por ejemplo, la instrucción

```
areaRect = AREA_RECTANGULO( a + 4, b + 7 );
```

se desarrolla como

```
areaRect = ( ( a + 4) * ( b + 7 ) );
```

El valor de la expresión se evalúa y se asigna a la variable **areaRect**.

Por lo general, el texto de reemplazo para la macro o la constante simbólica es cualquier texto en la línea después del identificador en la directiva **#define**. Si el texto de reemplazo para una macro o una constante simbólica es mayor que el resto de la línea, debe colocarse una *diagonal invertida* (\) al final de la línea, indicando que el texto de reemplazo continúa en la siguiente línea.

Las constantes simbólicas y las macros pueden descartarse mediante la *directiva de preprocesador* **#undef**. La directiva **#undef** "indefine" el nombre de una constante simbólica o de una macro. El *alcance* de una constante simbólica o de una macro es a partir de su definición y hasta su indefinición con **#undef**, o hasta el final del archivo. Una vez indefinido, puede definirse un nombre con **#define**.

Las funciones de la biblioteca algunas veces se definen como macros basadas en otras funciones de biblioteca. Una macro comúnmente definida en el encabezado **stdio.h** es

```
#define getchar() getc( stdin )
```

La definición de la macro **getchar** utiliza la función **getc** para obtener un carácter desde el flujo de entrada estándar. La función **putchar** del encabezado **stdio.h** y las funciones de manipulación de caracteres del encabezado **ctype.h** a menudo también se implementan como macros. Observe que las expresiones con efectos colaterales (es decir, que modifican los valores de las variables) no deben pasarse a una macro, debido a que los argumentos de una macro pueden evaluarse más de una vez.

## 13.5 Compilación condicional

La *compilación condicional* permite al programador controlar la ejecución de las directivas del preprocesador y la compilación del código de un programa. Cada una de las directivas condicionales del preprocesador eva-

lúa una expresión entera constante. Las expresiones de conversión de tipo, las expresiones **sizeof** y las constantes de enumeración no pueden evaluarse en las directivas del preprocesador.

La construcción condicional del preprocesador es similar a la instrucción de selección **if**. Considere el siguiente código de preprocesador:

```
#if !defined(NULL)
   #define NULL 0
#endif
```

Estas directivas determinan si **NULL** está definido. La expresión **defined(NULL)** da como resultado **1** si **NULL** está definido; de lo contrario devuelve **0**. Si el resultado es **0**, **!defined(NULL)** da como resultado **1** y se define **NULL**. De lo contrario, se ignora la directiva **#define**. Toda construcción **#if** termina con **#endif**. La directiva **#ifdef** y **#ifndef** son abreviaturas de **#if defined** (*nombre*) e **#if !defined**(*nombre*). Una construcción condicional de una directiva de varias partes puede evaluarse por medio de las directivas **#elif** (el equivalente de **else if** en una instrucción **if**) y **#else** (el equivalente de **else** en una instrucción **if**).

Durante el desarrollo de un programa, los programadores frecuentemente encuentran útil "comentar" porciones de código para evitar su compilación. Si el código contiene comentarios **/*** y ***/**, éste no podrá utilizarse para llevar a cabo su tarea. En su lugar, el programador puede utilizar la siguiente construcción de preprocesador:

```
#if 0
   código que no debe compilarse
#endif
```

Para permitir que el código se compile, remplace el **0** con **1** en la construcción anterior.

Con frecuencia, la compilación condicional se utiliza como un apoyo para la depuración. Muchas implementaciones de C proporcionan *depuradores*, los cuales brindan características mucho más poderosas que la compilación condicional. Si un depurador no está disponible, con frecuencia se utilizan instrucciones **printf** para imprimir los valores de las variables y para confirmar el flujo de control. Estas instrucciones **printf** pueden encerrarse dentro de directivas de preprocesador de modo que solamente se compilen mientras no termine el proceso de depuración. Por ejemplo,

```
#ifdef DEPURAR
   printf( "La variable x = %d\n", x );
endif
```

provoca que la instrucción **printf** se compile en el programa, si la constante simbólica **DEPURAR (#define DEPURAR)** se definió antes de la directiva **#ifdef DEPURAR**. Cuando termina la depuración, la directiva **#define** se elimina del archivo fuente, y las instrucciones **printf** insertadas para propósitos de depuración se ignoran durante la compilación. En programas más grandes podría ser recomendable definir varias constantes simbólicas diferentes que controlen la compilación condicional en secciones separadas del código fuente.



**Error común de programación 13.2**

*Insertar instrucciones* ***printf*** *compiladas condicionalmente para efectos de depuración en lugares donde C espera instrucciones individuales, es un error. En este caso, la instrucción compilada condicionalmente debe encerrarse en una instrucción compuesta. Así, cuando un programa se compile con instrucciones de depuración, el flujo de control del programa no se altera.*

## 13.6 Las directivas de preprocesador #error y #pragma

La *directiva* **#error**

```
#error tokens
```

imprime un mensaje que depende de la implementación, y que incluye los *tokens* especificados en la directiva. Los tokens son secuencias de caracteres separados por espacios. Por ejemplo:

```
#error 1 – Error fuera de rango
```

contiene 6 tokens. Cuando la directiva **#error** se procesa en algunos sistemas, los tokens en la directiva se despliegan como un mensaje de error, el procesamiento se detiene y el programa no se compila.

La *directiva* **#pragma**

**#pragma** *tokens*

provoca una acción definida por la implementación. Un pragma no reconocido por la implementación, se ignora. Para mayor información sobre **#error** y **#pragma**, vea la documentación correspondiente a su implementación de C.

## 13.7 Los operadores # y ##

Los operadores de preprocesador **#** y **##** están disponibles en el C estándar. El operador **#** provoca que un token del texto de reemplazo se convierta en una cadena encerrada entre comillas. Considere la siguiente definición de macro:

```
#define HOLA(x) printf( “Hola, ”  #x “\n” );
```

Cuando **HOLA(Juan)** aparece en un archivo del programa, ésta se desarrolla como

```
printf( “Hola, ”  “Juan” “\n” );
```

La cadena **“Juan”** reemplaza a **#x** en el texto de reemplazo. Las cadenas separadas por un espacio en blanco se concatenan durante el preprocesamiento, de manera que la instrucción es equivalente a

```
printf( “Hola, Juan\n”);
```

Observe que el operador **#** debe utilizarse en una macro con argumentos, ya que el operando de **#** hace referencia a un argumento de la macro.

El operador **##** concatena dos tokens. Considere la siguiente definición de macro:

```
#define CONCATTOKEN(x, y) x ## y
```

Cuando **CONCATTOKEN(x, y)** aparece en el programa, sus argumentos se concatenan y se utilizan para reemplazar la macro. Por ejemplo, **CONCATTOKEN(O, K)** se reemplaza con **OK** en el programa. El operador **##** debe tener dos operandos.

## 13.8 Números de línea

La *directiva de preprocesador* **#line** provoca que las líneas subsiguientes de código fuente se renumeren, comenzando con el valor entero constante especificado. La directiva

```
#line 100
```

comienza la numeración de líneas desde **100**, a partir de la siguiente línea de código fuente. Es posible incluir un nombre de archivo en la directiva **#line**. La directiva

```
#line 100 “archivo1.c”
```

indica que las líneas se numeran desde **100**, a partir de la siguiente línea de código, y que el nombre del archivo es **“archivo1.c”**, para efectos de mensajes del compilador. Por lo general, la directiva se utiliza para ayudar a que los mensajes producidos por errores de sintaxis y las advertencias del compilador sean más claros. Los números de línea no aparecen en el código fuente.

## 13.9 Constantes simbólicas predefinidas

El C de ANSI proporciona *constantes simbólicas predefinidas* (figura 13.1). Los identificadores para cada una de las constantes simbólicas predefinidas comienzan y terminan con *dos* guiones bajos. Estos identificadores y el identificador **defined** (utilizado en la sección 13.5) no pueden utilizarse en las directivas **#define** o **#undef**.

| Constante simbólica | Explicación |
|---|---|
| **__LINE__** | El número de línea del código fuente actual (una constante entera). |
| **__FILE__** | El nombre del archivo fuente (una cadena). |
| **__DATE__** | La fecha de compilación del código fuente (una cadena de la forma **"Mmm dd yyyy"**, tal como **"Jan 19 2002"**). |
| **__TIME__** | La hora de compilación de archivo fuente (una literal de cadena de la forma **"hh:mm:ss"**). |

**Figura 13.1** Algunas constantes simbólicas predefinidas.

## 13.10 Afirmaciones

La *macro* **assert**, definida en el encabezado **assert.h**, evalúa el valor de una expresión. Si el valor de la expresión es **0** (falso), **assert** imprime un mensaje de error y llama a la función **abort** (de la biblioteca general de utilidades, **stdlib.h**) para terminar la ejecución del programa. Por ejemplo, suponga que en un programa, la variable **x** nunca debe ser mayor que **10**. Es posible utilizar una afirmación para evaluar el valor de **x** e imprimir un mensaje de error si el valor de **x** es incorrecto. La instrucción sería

```
assert( x <= 10 );
```

Si **x** es mayor que **10** cuando el programa encuentra la instrucción anterior, se imprime un mensaje de error que contiene el número de línea y termina el programa. El programador puede entonces concentrarse en esa porción de código para encontrar el error. Si se define la constante simbólica **NDEPURAR**, las afirmaciones subsiguientes se ignoran. Así, cuado las afirmaciones ya no son necesarias, la línea

```
#define NDEPURAR
```

se inserta en el archivo del programa, en lugar de eliminar cada afirmación de forma manual.

## *RESUMEN*

- Todas las directivas de preprocesador comienzan con **#**.
- En una línea, solamente los caracteres blancos pueden aparecer antes de la directiva de preprocesador.
- La directiva **#include** incluye una copia del archivo especificado. Si el nombre del archivo se encierra entre comillas, el preprocesador comienza la búsqueda del archivo en el mismo directorio en donde se encuentra el archivo a compilar. Si el nombre del archivo se encierra entre llaves angulares (**<** y **>**), la búsqueda se realiza de la manera definida por la implementación.
- La directiva de preprocesador **#define** se utiliza para crear constantes simbólicas y macros.
- Una constante simbólica es el nombre de una constante.
- Una macro es una operación definida dentro de una directiva de preprocesador **#define**. Las macros pueden definirse con o sin argumentos.
- El texto de reemplazo para una macro o una constante simbólica es cualquier texto restante en la línea después del identificador de la directiva **#define**. Si el texto de reemplazo de una macro o una constante simbólica es mayor que el resto de la línea, se coloca una diagonal invertida (\) al final de la línea, indicando que el texto de reemplazo continúa en la siguiente línea.
- Las constantes simbólicas y las macros pueden descartarse por medio de la directiva de preprocesador **#undef**. La directiva de preprocesador **#undef** "indefine" el nombre de una constante simbólica o de una macro.
- El alcance de una constante simbólica o de una macro comienza en su definición y termina hasta su indefinición con **#undef**, o hasta el final del archivo.
- La compilación condicional permite al programador controlar la ejecución de las directivas de preprocesador y la compilación del código del programa.
- Las directivas de preprocesador condicionales evalúan expresiones constantes enteras. Las expresiones de conversión de tipo, las expresiones **sizeof** y las constantes de enumeración no pueden evaluarse dentro de las directivas de preprocesador.
- Cada construcción **#if** termina con **#endif**.

- Las directivas **#ifdef** e **#ifndef** son abreviaturas de **#if defined(** ***nombre*** **)** e **#if !defined(** ***nombre*** **)**.
- Las construcciones condicionales de varias partes del preprocesador pueden probarse por medio de las directivas **#elif** y **#else**.
- La directiva **#error** imprime un mensaje que depende de la implementación, el cual incluye los tokens especificados en la directiva.
- La directiva **#pragma** provoca una acción definida en la implementación. Si la implementación no reconoce el pragma, lo ignora.
- El operador **#** provoca que un token de texto de reemplazo se convierta en una cadena encerrada entre comillas. El operador **#** debe utilizarse en una macro con argumentos, debido a que el operando de **#** debe ser un argumento de la macro.
- El operador **##** concatena dos tokens. El operador **##** debe tener dos operandos.
- La directiva de preprocesador **#line** provoca que las líneas subsiguientes del código fuente se renumeren a partir del valor entero constante especificado.
- La constante **__LINE__** es el número de línea del código fuente actual (un entero). La constante **__FILE__** es el nombre del archivo (una cadena). La constante **__DATE__** es la fecha de compilación del código fuente (una cadena). La constante **__TIME__** es la hora de compilación del código fuente (una cadena). Observe que cada una de las constantes simbólicas predefinidas comienza y termina con dos guiones bajos.
- La macro **assert**, definida en el encabezado **assert.h**, evalúa el valor de una expresión. Si el valor de la expresión es **0** (falso), **assert** imprime un mensaje de error y llama a la función **abort** para terminar la ejecución del programa.

## TERMINOLOGÍA

\ (diagonal invertida) carácter de continuación
**abort**
alcance de una constante simbólica o macro
argumento
**assert**
**assert.h**
compilación condicional
constante simbólica
constantes simbólicas predefinidas
**__DATE__**
**#define**
depurador
desarrollar una macro
directiva de preprocesador
ejecución condicional de directivas de preprocesador
**#elif**
**#else**
encabezados de la biblioteca estándar
**#endif**
**#error**
**__FILE__**
**#if**
**#ifdef**
**#ifndef**
**#include "<nombre de archivo"**
**#include <nombre de archivo>**
**#line**
**__LINE__**
macro
macro con argumentos
operador **#** de conversión a cadenas
operador **##** de concatenación
**#pragma**
preprocesador de C
**stdio.h**
**stdlib.h**
texto de reemplazo
**__TIME__**
**#undef**

## ERRORES COMUNES DE PROGRAMACIÓN

**13.1** Olvidar encerrar los argumentos de una macro entre paréntesis en el texto de reemplazo, puede provocar errores de lógica.

**13.2** Insertar instrucciones **printf** compiladas condicionalmente para efectos de depuración en lugares donde C espera instrucciones individuales, es un error. En este caso, la instrucción compilada condicionalmente debe encerrarse en una instrucción compuesta. Así, cuando un programa se compile con instrucciones de depuración, el flujo de control del programa no se altera.

## BUENA PRÁCTICA DE PROGRAMACIÓN

**13.1** Utilizar nombres significativos para las constantes simbólicas ayuda a hacer programas más autodocumentados.

## TIP DE RENDIMIENTO

**13.1** Algunas veces, las macros pueden utilizarse para reemplazar una llamada a una función con código **inline** antes del tiempo de ejecución. Esto elimina la sobrecarga de llamadas a la función.

## EJERCICIOS DE AUTOEVALUACIÓN

**13.1** Complete los espacios en blanco:

a) Toda directiva de preprocesador debe comenzar con ____________.
b) La estructura de compilación condicional debe desarrollarse para evaluar distintos casos por medio de las directivas ____________ y ____________.
c) La directiva ____________ crea macros y constantes simbólicas.
d) En una línea, sólo los caracteres ____________ pueden aparecer antes de una directiva de preprocesador.
e) La directiva ____________ descarta la constante simbólica y los nombres de las macros.
f) Las directivas ____________ y ____________ se proporcionan como una abreviatura de **`#ifdefined (`*`nombre`*`)`** y de **`#if!defined(`*`nombre`*`)`**.
g) ____________ permite al programador controlar la ejecución de las directivas del preprocesador y la ejecución del código del programa.
h) La macro ____________ imprime un mensaje y termina la ejecución del programa, si el valor de la expresión que la macro evalúa es **`0`**.
i) La directiva ____________ inserta un archivo en otro archivo.
j) El operador ____________ concatena dos argumentos.
k) El operador ____________ convierte su operando en una cadena.
l) El carácter ____________ indica que el texto de reemplazo para una constante simbólica o una macro continúa en la siguiente línea.
m) La directiva ____________ provoca la numeración de las líneas de código fuente desde el valor indicado, a partir de la siguiente línea de código.

**13.2** Escriba un programa que imprima los valores de las constantes simbólicas listadas en la figura 13.1.

**13.3** Escriba una directiva de preprocesador para llevar a cabo cada una de las siguientes tareas:

a) Defina la constante simbólica **`SI`** que tenga un valor igual a **`1`**.
b) Defina la constante simbólica **`NO`** que tenga un valor igual a **`0`**.
c) Incluya el encabezado **`comun.h`**. El encabezado se encuentra en el mismo directorio en donde se encuentra el archivo que va a compilarse.
d) Renumere las líneas restantes del archivo a partir de **`3000`**.
e) Si la constante simbólica **`VERDADERO`** está definida, indefínala y redefínala como **`1`**. No utilice **`#ifdef`**.
f) Si la constante simbólica **`VERDADERO`** está definida, indefínala y redefínala como **`1`**. Utilice **`#ifdef`**.
g) Si la constante simbólica **`VERDADERO`** no es igual que **`0`**, defina la constante simbólica **`FALSO`** como **`0`**. De lo contrario defina **`FALSO`** igual a **`1`**.
h) Defina la macro **`VOLUMEN_CUADRADO`** que calcule el volumen de un cuadrado. La macro toma un argumento.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**13.1** a) **`#`**. b) **`#elif`**, **`#else`**. c) **`#define`**. d) Blancos. e) **`#undef`**. f) **`#ifdef`**, **`#ifndef`**. g) Compilación condicional. h) **`assert`**. i) **`#include`**. j) **`##`**. k) **`#`**. l) **`\`** m) **`#line`**.

**13.2**

```
/* Imprime los valores de las macros predefinidas */
#include <stdio.h>
int main()
{
   printf( "_LINE_ = %d\n", _LINE_ );
   printf( "_FILE_ = %d\n", _LINE_ );
   printf( "_DATE_ = %d\n", _LINE_ );
   printf( "_TIME_ = %d\n", _LINE_ );
   return 0;
}
```

```
_LINE_ = 5
_FILE_ = macros.c
_DATE_ = Jun  5 2003
_TIME_ = 09:38:58
```

**13.3** a) `#define SI 1`
b) `#define NO 0`
c) `#include “comun.h”`
d) `#line 3000`
e)
```
#if defined(VERDADERO)
    #undef VERDADERO
    #define VERDADERO 1
    #endif
```
f)
```
#ifdef VERDADERO
    #undef VERDADERO
    #define VERDADERO 1
#endif
```
g)
```
#if VERDADERO
    #define FALSO 0
#else
    #define FALSO 1
    #endif
```
h) `#define VOLUMEN_CUADRADO( x ) ( x )*( x ) * ( x )`

## EJERCICIOS

**13.4** Escriba un programa que defina una macro con un argumento para calcular el volumen de una esfera. El programa debe calcular el volumen para esferas con radios de 1 a 10 e imprimir los resultados en formato tabular. La fórmula para el volumen de la esfera es:

$$( 4.0 / 3 ) * \pi * r^3$$

donde π es **3.14159**.

**13.5** Escriba un programa que produzca la siguiente salida:
La suma de **x** y **y** es **13**
El programa debe definir la macro **SUMA** con dos argumentos, **x** y **y**, y utilizar **SUMA** para producir la salida.

**13.6** Escriba un programa que defina y utilice la macro **MINIMO2** para determinar el más pequeño de dos valores numéricos. Introduzca los valores desde el teclado.

**13.7** Escriba un programa que defina y utilice la macro **MINIMO3** para determinar el más pequeño de tres valores numéricos. La macro **MINIMO3** debe utilizar la macro **MINIMO2** definida en el ejercicio 13.6 para determinar el valor más pequeño. Introduzca los valores desde el teclado.

**13.8** Escriba un programa que defina y utilice la macro **IMPRIME** para imprimir un valor de cadena.

**13.9** Escriba un programa que defina y utilice la macro **IMPRIMEARREGLO** para imprimir un arreglo de enteros. La macro debe recibir como argumentos, el arreglo y el número de elementos en el arreglo.

**13.10** Escriba un programa que defina y utilice la macro **SUMAARREGLO** para sumar los valores de un arreglo numérico. La macro debe recibir como argumentos, el arreglo y el número de elementos en el arreglo.

# 14

# Otros temas de C




## Objetivos

- Redireccionar la entrada del teclado para que provenga desde un archivo.
- Redireccionar la salida de la pantalla para que se coloque en un archivo.
- Escribir funciones que utilicen listas de argumentos de longitud variable.
- Procesar argumentos de línea de comandos.
- Asignar tipos específicos a constantes numéricas.
- Utilizar archivos temporales.
- Procesar eventos inesperados dentro de un programa.
- Asignar memoria de manera dinámica para arreglos.
- Modificar el tamaño de la memoria previamente asignada de manera dinámica.

*Utilizaremos una señal que he probado y que he encontrado muy fácil de gritar. ¡Waa-hoo!*
Zane Grey

*Utilízalo, póntelo.*
*Haz que lo haga, o hazlo sin él.*
Anónimo

*Es un problema de tres rutas.*
Sir Arthur Conan Doyle

## Plan general



## 14.1 Introducción

Este capítulo presenta varios temas adicionales que por lo general no se explican en cursos de introducción. Muchas de las capacidades que explicamos aquí son específicas de sistemas operativos particulares, especialmente UNIX y Windows.

## 14.2 Cómo redireccionar la entrada/salida en sistemas UNIX y Windows

Por lo general, la entrada hacia un programa es desde el teclado (entrada estándar), y la salida es hacia la pantalla (salida estándar). En la mayoría de los sistemas de cómputo, en especial en los sistemas UNIX y Windows, es posible *redireccionar* las entradas para que provengan desde un archivo en lugar de hacerlo desde el teclado, y redireccionar la salida para que se coloque en un archivo y no en la pantalla. Ambas formas de redirección se pueden llevar a cabo sin utilizar las capacidades de procesamiento de archivos de la biblioteca estándar.

Existen varias maneras de redireccionar la entrada y la salida desde la línea de comandos de UNIX. Considere el archivo ejecutable **`suma`** que introduce enteros uno a uno, que mantiene la suma total de los valores hasta que se establece el indicador de fin de archivo, y luego imprime el resultado. Por lo general, el usuario introduce los enteros desde el teclado e introduce la combinación de teclas de fin de archivo para indicar que no introducirá más valores. Con la redirección de la entrada, ésta puede almacenarse en un archivo. Por ejemplo, si los datos se almacenan en el archivo entrada, la línea de comando

```
$ suma < entrada
```

ejecuta el programa **`suma`**; el *símbolo de redirección de entrada (*`<`*)* indica que el archivo de datos entrada se utilizará como entrada para el programa. La redirección de la entrada en un sistema Windows es idéntica.

Observe que **`$`** es un indicador de línea de comando de UNIX (algunos sistemas UNIX utilizan el indicador **`%`**, u otro símbolo). Con frecuencia, a los estudiantes se les dificulta comprender que la redirección es una función del sistema operativo, y no otra característica de C.

El segundo método para redireccionar la entrada es la *canalización*. Una *canalización* (**`|`**) provoca que la salida de un programa se redireccione como la entrada de otro programa. Suponga que el programa aleatorio despliega una serie de enteros al azar; la salida de aleatorio puede "canalizarse" directamente hacia el programa **`suma`** por medio de la línea de comandos de UNIX

```
$ aleatorio | suma
```

Esto provoca que se calcule la suma de los enteros producidos por **aleatorio**. La canalización se realiza de manera idéntica en UNIX y en Windows.

La salida de un programa puede redireccionarse hacia un archivo por medio del *símbolo de redirección de salida* (**>**) (se utiliza el mismo símbolo en UNIX y en Windows). Por ejemplo, para redireccionar la salida del programa **aleatorio** hacia el archivo **fuera**, utilice

```
$ aleatorio > fuera
```

Por último, la salida del programa puede agregarse al final de un archivo existente utilizando el *símbolo de agregar a la salida* (**>>**) (se utiliza el mismo símbolo en UNIX y en Windows). Por ejemplo, para agregar la salida del programa **aleatorio** al archivo **fuera** creado en la línea de comando anterior, utilice la línea de comando

```
$ aleatorio >> fuera
```

## 14.3 Listas de argumentos de longitud variable

Es posible crear funciones que reciban un número no especificado de argumentos. La mayoría de los programas de este libro utilizan la función **printf** de la biblioteca estándar, la cual, como usted sabe, toma un número variable de argumentos. Como mínimo, **printf** debe recibir una cadena como primer argumento, pero **printf** puede recibir cualquier número adicional de argumentos. El prototipo de la función es

```
int printf( const char *formato, . . . );
```

Los *puntos suspensivos* (**. . .**) en el prototipo de la función indican que la función recibe un número variable de argumentos de cualquier tipo. Observe que los puntos suspensivos siempre deben colocarse al final de la lista de parámetros.

Las macros y las definiciones de los *encabezados de argumentos variables* ***stdarg.h*** (figura 14.1) proporcionan las capacidades necesarias para construir funciones con *listas de argumentos de longitud variable*. La figura 14.2 muestra la función **promedio** (línea 28), la cual recibe un número variable de argumentos. El primer argumento de la función **promedio** siempre es el número de valores a promediar.

| Identificador | Explicación |
|---|---|
| **va_list** | Tipo que puede personalizarse para almacenar información necesaria para las macros **va_start**, **va_arg** y **va_end**. Para acceder a los argumentos de una lista de argumentos de longitud variable, debe definirse un objeto de tipo **va_list**. |
| **va_start** | Macro que se invoca antes de poder acceder a los argumentos de la lista de argumentos de longitud variable. La macro inicializa el objeto declarado con **va_list** para que pueda utilizarse con las macros **va_arg** y **va_end**. |
| **va_arg** | Macro que se amplía a una expresión del valor y tipo del siguiente argumento de la lista de argumentos de longitud variable. Cada invocación de **va_arg** modifica el objeto declarado con **va_list**, de modo que el objeto apunte al siguiente argumento en la lista. |
| **va_end** | Macro que facilita un retorno normal desde una función a cuya lista de argumentos de longitud variable se hizo referencia por medio de la macro **va_start**. |

**Figura 14.1** Tipos y macros de la lista de argumentos de longitud variable de **stdarg.h**.

```
1  /* Figura 14.2: fig14_02.c
2     Uso de listas de argumentos de longitud variable*/
3  #include <stdio.h>
4  #include <stdarg.h>
```

**Figura 14.2** Uso de listas de argumentos de longitud variable. (Parte 1 de 2.)

```
5
6  double promedio( int i, ... ); /* prototipo */
7
8  int main()
9  {
10    double w = 37.5;
11    double x = 22.5;
12    double y = 1.7;
13    double z = 10.2;
14
15    printf( “%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n”,
16       “w = “, w, “x = “, x, “y = “, y, “z = “, z );
17    printf( “%s%.3f\n%s%.3f\n%s%.3f\n”,
18       “El promedio de w y x es “, promedio( 2, w, x ),
19       “El promedio de w, x, y y es “, promedio( 3, w, x, y ),
20       “El promedio de w, x, y, y z es “,
21       promedio( 4, w, x, y, z ) );
22
23    return 0; /* indica terminación exitosa */
24
25 } /* fin de main */
26
27 /* calcula el promedio */
28 double promedio( int i, ... )
29 {
30    double total = 0; /* inicializa el total */
31    int j; /* contador para seleccionar argumentos */
32    va_list ap; /* almacena la información necesaria para va_start y va_end */
33
34    va_start( ap, i ); /* inicializa el objeto va_list */
35
36    /* procesa la lista de argumentos de longitud variable */
37    for ( j = 1; j <= i; j++ ) {
38       total += va_arg( ap, double );
39    } /* fin de for */
40
41    va_end( ap ); /* limpia la lista de argumentos de longitud variable */
42
43    return total / i; /* calcula el promedio */
44 } /* fin de la función promedio */
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

El promedio de w y x es 30.000
El promedio de w, x, y y es 20.567
El promedio de w, x, y, y z es 17.975
```

**Figura 14.2** Uso de listas de argumentos de longitud variable. (Parte 2 de 2.)

La función **`promedio`** (líneas 20 a 44) utiliza todas las definiciones y macros del encabezado **`stdarg.h`**. Las macros **`va_start`**, **`va_arg`** y **`va_end`** utilizan el objeto **`ap`** de tipo **`va_list`** (línea 32), para procesar la lista de argumentos de longitud variable de la función **`promedio`**. La función comienza invocando a la ma-

cro **va_start** (línea 34) para inicializar el objeto **ap** y utilizarlo con **va_arg** y **va_end**. La macro recibe dos argumentos, el objeto **ap** y el identificador más a la derecha de la lista de argumentos antes de los puntos suspensivos, en este caso **i** (**va_start** utiliza **i** para determinar en dónde comienza la lista de argumentos de longitud variable). A continuación, la función **promedio** suma de manera repetida los argumentos de la lista de argumentos de longitud variable a **total** (líneas 37 a 39). El valor a sumarse a **total** se recupera desde la lista de argumentos al invocar a la macro **va_arg**. La macro **va_arg** recibe dos argumentos, el objeto **ap** y el tipo de valor esperado en la lista de argumentos, que en este caso son **double**. La macro devuelve el valor del argumento. La función **promedio** invoca a la macro **va_end** (línea 41) con el objeto **ap** como un argumento, para facilitar un retorno normal a **main** desde **promedio**. Por último, se calcula el promedio y se devuelve a **main**.



**Error común de programación 14.1**

*Colocar puntos suspensivos en medio de la lista de parámetros de una función, es un error de sintaxis. Los puntos suspensivos solamente pueden colocarse al final de la lista de parámetros.*

El lector podría preguntarse cómo es que las funciones **printf** y **scanf** saben qué tipo utilizar en cada macro **va_arg**. La respuesta es que **printf** y **scanf** exploran los especificadores en la cadena de control de formato para determinar el tipo del siguiente argumento a procesar.

## 14.4 Uso de argumentos en la línea de comandos

En muchos sistemas, es posible pasar argumentos hacia **main** desde la línea de comandos, si se incluyen los parámetros **int argc** y **char *argv[]** en la lista de parámetros de **main**. El parámetro **argc** recibe el número de argumento de la línea de comandos. El parámetro **argv** es un arreglo de cadenas en el que se almacenan los parámetros de la línea de comandos. Los usos más comunes de los argumentos en la línea de comandos incluye la impresión de argumentos, el paso de opciones a un programa y el paso de nombres de archivo a un programa.

La figura 14.3 copia un archivo a otro, carácter por carácter. Asumimos que el archivo ejecutable del programa se llama **miCopia**. Una línea de comandos común para el programa **miCopia** en los sistemas UNIX es

```
$ miCopia entrada salida
```

Esta línea de comandos indica que el archivo **entrada** va a copiarse en el archivo **salida**. Cuando se ejecuta el programa, si **argc** no es **3** (**miCopia** cuenta como un argumento), el programa imprime un mensaje de error y termina. De lo contrario, el arreglo **argv** almacena las cadenas **"miCopia"**, **"entrada"** y **"salida"**. El programa utiliza el segundo y tercer argumento de la línea de comandos como los nombres de los archivos. Los archivos se abren por medio de la función **fopen**. Si ambos archivos se abren con éxito, los caracteres se leen desde el archivo **entrada** y se escriben en el archivo **salida**, hasta encontrar el indicador de fin de archivo en el archivo **entrada**. Después, el programa termina. El resultado es una copia exacta de **entrada**. Revise los manuales de su sistema para mayor información acerca de los argumentos en la línea de comandos.

```
/* Figura 14.3: fig14_03.c
   Uso de argumentos en la línea de comandos */
#include <stdio.h>

int main( int argc, char *argv[] )
{
   FILE *ptrEntArchivo;  /* apuntador de archivo de entrada */
   FILE *ptrSalArchivo;  /* apuntador de archivo de salida */
   int c;                /* define c para almacenar los caracteres
                            introducidos por el usuario */

```

**Figura 14.3** Uso de argumentos en la línea de comandos. (Parte 1 de 2.)

```
   /* verifica el número de argumentos de la línea de comandos */
   if ( argc != 3 ) {
      printf( "Uso: copia archivoEnt archivoSal\n" );
   } /* fin de if */
   else {

      /* si el archivo de entrada se puede abrir */
      if ( ( ptrEntArchivo = fopen( argv[ 1 ], "r" ) ) != NULL ) {

         /* si el archivo de salida se puede abrir */
         if ( ( ptrSalArchivo = fopen( argv[ 2 ], "w" ) ) != NULL ) {

            /* lee los caracteres y los arroja */
            while ( ( c = fgetc( ptrEntArchivo ) ) != EOF ) {
               fputc( c, ptrSalArchivo );
            } /* fin de while */

         } /* fin de if */
         else { /* no se puede abrir el archivo de salida */
            printf( "El archivo \"%s\" no se pudo abrir\n", argv[ 2 ] );
         } /* fin de else */

      } /* fin de if */
      else { /* no se puede abrir el archivo de entrada */
         printf( "El archivo \"%s\" no se pudo abrir\n", argv[ 1 ] );
      } /* fin de else */

   } /* fin de else */

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

**Figura 14.3** Uso de argumentos en la línea de comandos. (Parte 2 de 2.)

## 14.5 Notas sobre la compilación de programas con múltiples archivos fuente

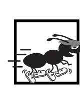
Como establecimos anteriormente en el libro, es posible construir programas que consten de múltiples archivos fuente (vea el capítulo 16). Existen varias cuestiones que deben tomarse en cuenta cuando se generen programas en múltiples archivos. Por ejemplo, la definición de una función debe estar completamente en un archivo, no se puede dividir en dos o más archivos.

En el capítulo 5, introdujimos los conceptos de clase de almacenamiento y alcance. Aprendimos que las variables declaradas fuera de cualquier definición de una función tienen de manera predeterminada una clase de almacenamiento **`static`** y se les denomina variables globales. Las variables globales son accesibles para cualquier función definida en el mismo archivo después de la declaración de la variable. Las variables globales también son accesibles a funciones en otros archivos. Sin embargo, las variables globales deben declararse en cada archivo en donde se utilicen. Por ejemplo, si definimos la variable global entera **`bandera`** en un archivo y hacemos referencia a ella en otro archivo, el segundo archivo debe contener la declaración

```
extern int bandera;
```

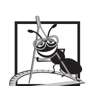
antes de utilizar la variable en dicho archivo. Esta declaración utiliza el especificador de clase de almacenamiento **`extern`** para indicar que la variable **`bandera`** se define más adelante en el mismo archivo o en un archivo diferente. El compilador informa al enlazador que aparecen referencias no resueltas hacia la variable **`bandera`** dentro del archivo (el compilador no sabe en dónde está definida **`bandera`**, de modo que permite

que el enlazador intente encontrar **bandera**). Si el enlazador no puede localizar una definición para **bandera**, lanza un mensaje de error y no produce un archivo ejecutable. Si el enlazador encuentra una definición global apropiada, resuelve la referencia indicando en dónde se localiza **bandera**.



### Tip de rendimiento 14.1

*Las variables globales incrementan el rendimiento debido a que se puede acceder a ellas directamente desde cualquier función, y se elimina la sobrecarga de pasar datos a funciones.*



### Observación de ingeniería de software 14.1

*Las variables globales deben evitarse, a menos que sean indispensables para el rendimiento de la aplicación, ya que éstas violan el principio del menor privilegio.*

Tal como las declaraciones **extern** pueden utilizarse para declarar variables globales en otros archivos de programa, los prototipos de las funciones pueden ampliar el alcance de una función más allá del archivo en el que se definió (el especificador **extern** no se requiere en el prototipo de la función). Esto se lleva a cabo incluyendo el prototipo de la función en cada archivo en el que se invoque a la función, y compilando juntos a los archivos (vea la sección 13.2). Los prototipos de las funciones indican al compilador que la función especificada está definida, ya sea más adelante en el mismo documento, o en un archivo diferente. De nuevo, el compilador no intenta resolver las referencias a dichas funciones, esa tarea se la deja al enlazador. Si el enlazador no puede localizar una definición de función apropiada, éste emite un mensaje de error.

Para ejemplificar el uso de los prototipos de función para ampliar el alcance de una función, considere cualquier programa que contenga la directiva de preprocesador **#include <stdio.h>**. Esta directiva incluye en un archivo los prototipos para funciones tales como **printf** y **scanf**. Otras funciones en el archivo pueden utilizar **printf** y **scanf** para llevar a cabo sus tareas. Las funciones **printf** y **scanf** se definen en otros archivos. No necesitamos saber en dónde están definidas. Simplemente reutilizamos el código dentro de nuestro programa. El enlazador resuelve automáticamente nuestras referencias a estas funciones. Este proceso permite utilizar las funciones de la biblioteca estándar.



### Observación de ingeniería de software 14.2

*Crear programas en distintos archivos fuente facilita la reutilización de software y la buena ingeniería de software. Las funciones pueden ser comunes a muchas aplicaciones. En dichas circunstancias esos archivos tienen que almacenarse en sus propios archivos fuente, y cada archivo fuente debe tener el archivo de encabezado correspondiente que contenga los prototipos de las funciones. Esto permite a los programadores de diferentes aplicaciones reutilizar el mismo código, mediante la inclusión y compilación del archivo de encabezado apropiado para sus aplicaciones con el archivo fuente correspondiente.*

Es posible restringir el alcance de una variable global o de una función al archivo en el que están definidas. El especificador de clase de almacenamiento **static**, cuando se aplica a una variable global o a una función, evita que la utilice alguna función que no está definida en el mismo archivo. A esto se le conoce como *vinculación interna*. Las variables y las funciones globales que no son precedidas por **static** en sus definiciones tienen una *vinculación externa*; se puede acceder a ellas desde otros archivo, si dichos archivos contienen las declaraciones apropiadas y/o los prototipos de las funciones.

La declaración de la variable global

```
static double pi = 3.14159;
```

crea la variable **pi** de tipo **double**, la inicializa en **3.14159**, e indica que a **pi** solamente se le conoce en las funciones que están dentro del archivo en donde está definida.

Por lo general, el especificador **static** se utiliza con las funciones de utilidad que son llamadas por las funciones de un archivo en particular. Si no se requiere una función fuera de un archivo en particular, debe reforzarse el principio del menor privilegio por medio de **static**. Si una función se define antes de que se utilice en un archivo, **static** debe aplicarse en la definición de la función; de lo contrario, debe aplicarse al prototipo de la función.

Cuando se construyen programas grandes en múltiples archivos fuente, la compilación del programa se vuelve una tarea tediosa, si se hacen pequeños cambios a un archivo y debe compilarse el programa completo. Muchos sistemas proporcionan utilidades especiales que recompilan solamente el programa modificado. En sistemas UNIX a la utilidad se le llama **make**. La utilidad **make** lee un archivo llamado **makefile** que con-

tiene instrucciones para la compilación y el enlace del programa. Los productos tales como el C++ Builder de Borland y Visual C++ de Microsoft también proporcionan utilidades similares. Para mayor información respecto a las utilidades **make**, vea el manual correspondiente a su herramienta de desarrollo.

## 14.6 Terminación de un programa mediante exit y atexit

La biblioteca general de utilidades (**stdlib.h**) proporciona métodos para terminar la ejecución de los programas por medios no convencionales, como el retorno de la función **main**. La función **exit** fuerza la terminación exitosa de un programa, como si se ejecutara normalmente. Con frecuencia, la función se utiliza para terminar el programa cuando se detecta un error en la entrada, o si no se puede abrir un archivo que va a procesar el programa. La función **atexit** *registra* una función que debe llamarse durante la terminación exitosa de un programa, es decir, ya sea cuando el programa termina al llegar al final de **main**, o cuando se invoca a **exit**.

La función **atexit** toma como argumento un apuntador a una función (es decir, el nombre de la función). Las funciones llamadas durante la terminación del programa no pueden tener argumentos y no pueden devolver valor alguno. Se pueden registrar hasta 32 funciones para su ejecución en el momento de la terminación del programa.

La función **exit** toma un argumento. Por lo general, el argumento es una constante simbólica **EXIT_SUCCESS** o la constante simbólica **EXIT_FAILURE**. Si se llama a **exit** con **EXIT_SUCCESS**, ésta devuelve al ambiente que hizo la llamada el valor definido por la implementación para una terminación exitosa. Si se llama a **exit** con **EXIT_FAILURE**, ésta devuelve el valor para una terminación no exitosa, definida por la implementación. Cuando se invoca a la función **exit**, se invoca cualquier función previamente registrada con **atexit** en el orden inverso al de su registro, se depuran y se cierran todos los flujos asociados con el programa, y el control regresa al ambiente anfitrión. La figura 14.4 prueba las funciones **exit** y **atexit**. El programa indica al usuario que determine si el programa terminó con **exit**, o al alcanzar el final de **main**. Observe que la función imprime se ejecuta en cada caso, al terminar el programa.

```
/* Figura 14.4: fig14_04.c
   Uso de las funciones exit y atexit */
#include <stdio.h>
#include <stdlib.h>

void imprime( void ); /* prototipo */

int main()
{
   int respuesta; /* elección de menú del usuario */

   atexit( imprime ); /* registra la nueva función imprime */
   printf( “Introduzca 1 para terminar el programa con la funcion exit”
      “\nIntroduzca 2 para terminar el programa de manera normal\n” );
   scanf( “%d”, &respuesta );

   /* llama a exit si la respuesta es 1 */
   if ( respuesta == 1 ) {
      printf( “\nTermina el programa con la funcion exit\n” );
      exit( EXIT_SUCCESS );
   } /* fin de if */

   printf( “\nTermina el programa al encontrar el final de main\n” );

   return 0; /* indica terminación exitosa */
```

**Figura 14.4** Funciones **exit** y **atexit**. (Parte 1 de 2.)

```

} /* fin de main */

/* despliega un mensaje antes de terminar */
void imprime( void )
{
   printf( "Ejecuta la funcion imprime al "
       "finalizar el programa\n" );
} /* fin de la función imprime */
```

```
Introduzca 1 para terminar el programa con la funcion exit
Introduzca 2 para terminar el programa de manera normal
1

Termina el programa con la funcion exit
Ejecuta la funcion imprime al finalizar el programa
Programa terminado
```

```
Introduzca 1 para terminar el programa con la funcion exit
Introduzca 2 para terminar el programa de manera normal
2

Termina el programa al encontrar el final de main
Ejecuta la funcion imprime al finalizar el programa
Programa terminado
```

**Figura 14.4** Funciones **exit** y **atexit**. (Parte 2 de 2.)

## 14.7 El calificador de tipo volatile

En los capítulos 6 y 7, presentamos el calificador de tipo **const**. C también proporciona el calificador de tipo **volatile** para suprimir distintos tipos de optimización. El C estándar indica que cuando se utiliza **volatile** para calificar un tipo, la naturaleza del acceso a un objeto de ese tipo depende de la implementación.

## 14.8 Sufijos para las constantes enteras y de punto flotante

C proporciona sufijos enteros y de punto flotante para especificar las constantes de tipo entero y de punto flotante. Los sufijos enteros son: **u** y **U** para entero **unsigned**, **l** o **L** para entero **long**, y **ul**, **lu**, **UL** o **LU** para un entero **unsigned long**. Las siguientes constantes son de tipo **unsigned**, **long** y **unsigned long**, respectivamente:

```
174u
8358L
28373ul
```

Si una constante entera no tiene sufijo, su tipo se determina por medio del primer tipo capaz de almacenar un valor de ese tamaño (primero **int**, después **long int**, después **unsigned long int**).

Los sufijos de punto flotante son: **f** o **F** para punto **flotante**, y **l** o **L** para **long double**. Las siguientes constantes son de tipo **float** y **long double**, respectivamente:

```
1.28f
3.14159L
```

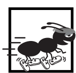
Una constante de punto flotante que no tiene sufijo es automáticamente de tipo **double**.

| Modo | Descripción |
|---|---|
| `rb` | Abre un archivo binario para lectura. |
| `wb` | Crea un archivo binario para escritura. Si el archivo ya existe, descarta el contenido actual. |
| `ab` | Agrega: abre o crea un archivo binario para escritura al final del archivo. |
| `rb+` | Abre un archivo binario para actualización (lectura y escritura). |
| `wb+` | Crea un archivo binario para actualización. Si el archivo ya existe, descarta su contenido actual. |
| `ab+` | Agrega: abre o crea un archivo binario para actualización; toda la escritura se hace al final del archivo. |

**Figura 14.5** Modos de apertura de un archivo binario.

## 14.9 Más acerca de los archivos

En el capítulo 11 presentamos las capacidades para procesamiento de archivos de texto con acceso secuencial y aleatorio. C también proporciona capacidades para el procesamiento de archivos binarios, pero algunos sistemas de cómputo no soportan archivos binarios. Si los archivos binarios no son soportados y el archivo se abre en modo de archivo binario (figura 14.5), el archivo se procesará como un archivo de texto. Los archivos binarios deben utilizarse en lugar de los archivos de texto, sólo en situaciones en donde la rigidez de velocidad, las condiciones de almacenamiento y/o compatibilidad requieren de archivos binarios. De lo contrario, los archivos de texto siempre son preferibles, debido a su portabilidad inherente y por la habilidad de utilizar otras herramientas estándar para examinar y manipular los archivos de datos.



**Tip de rendimiento 14.2**

*Considere utilizar archivos binarios en lugar de archivos de texto, en aplicaciones que demandan alto rendimiento.*



**Tip de portabilidad 14.1**

*Utilice archivos de texto, cuando escriba programas portables.*

Además de las funciones de procesamiento de archivos que explicamos en el capítulo 11, la biblioteca estándar proporciona la función **`tmpfile`** que abre un archivo temporal en modo **`"wb+"`**. Aunque éste es un modo de archivo binario, algunos sistemas procesan archivos temporales como archivos de texto. Un archivo temporal existe hasta que se cierra con **`fclose`**, o hasta que el programa termina.

La figura 14.6 cambia los tabuladores de un archivo por espacios. El programa indica al usuario que introduzca el nombre de un archivo a modificar. Si el archivo introducido por el usuario y el archivo temporal se abren con éxito, el programa lee los caracteres del archivo a modificar y los escribe en el archivo temporal. Si el carácter que lee es un tabulador (**`'\t'`**), lo reemplaza con un espacio y lo escribe en el archivo temporal. Cuando alcanza el fin de archivo, los apuntadores para cada archivo se reposicionan al principio de cada archivo mediante **`rewind`**. A continuación, el archivo temporal se copia dentro del archivo original, carácter por carácter. El programa imprime el archivo original mientras copia los caracteres dentro del archivo temporal e imprime el nuevo archivo mientras copia los caracteres desde el archivo temporal hacia el archivo original, para confirmar la escritura de los caracteres.

```
/* Figura 14.6: fig14_06.c
   Uso de archivos temporales */
#include <stdio.h>

int main()
{
   FILE *ptrArchivo;      /* apuntador al archivo a modificar */
```

**Figura 14.6** Archivos temporales. (Parte 1 de 2.)

```
    FILE *ptrArchivoTemp; /* apuntador al archivo temporal */
    int c; /* define c para almacenar los caracteres leídos desde el archivo */
    char nombreArchivo[ 30 ]; /* crea un arreglo de caracteres */

    printf( "Este programa cambia tabuladores por espacios.\n"
       "Introduzca un archivo a modificar: " );
    scanf( "%29s", nombreArchivo );

    /* fopen abre el archivo */
    if ( ( ptrArchivo = fopen( nombreArchivo, "r+" ) ) != NULL ) {

       /* crea un archivo temporal */
       if ( ( ptrArchivoTemp = tmpfile() ) != NULL ) {
          printf( "\nEl archivo antes de la modificacion es:\n" );

          /* lee caracteres desde un archivo y los coloca en un archivo temporal */
          while ( ( c = getc( ptrArchivo ) ) != EOF ) {
             putchar( c );
             putc( c == '\t' ? ' ': c, ptrArchivoTemp );
          } /* fin de while */

          rewind( ptrArchivoTemp );
          rewind( ptrArchivo );
          printf( "\n\nEl archivo despues de la modificacion es:\n" );

          /* lee desde un archivo temporal y escribe en el archivo original */
          while ( ( c = getc( ptrArchivoTemp ) ) != EOF ) {
             putchar( c );
             putc( c, ptrArchivo );
          } /* fin de while */

       } /* fin de if */
       else { /* si no se puede abrir el archivo temporal */
          printf( "No se puede abrir el archivo temporal\n" );
       } /* fin de else */

    } /* fin de if */
    else { /* si no se puede abrir el archivo */
       printf( "No se puede abrir el archivo %s\n", nombreArchivo );
    } /* fin de else */

    return 0; /* indica terminación exitosa */

} /* fin de main */
```

```
Este programa cambia tabuladores por espacios.
Introduzca un archivo a modificar: datos.txt

El archivo antes de la modificacion es:
0       1       2       3       4
        5       6       7       8       9

El archivo despues de la modificacion es:
0 1 2 3 4
 5 6 7 8 9
```

**Figura 14.6** Archivos temporales. (Parte 2 de 2.)

| Señal | Explicación |
|---|---|
| **SIGABRT** | Terminación anormal del programa (tal como una llamada a la función **abort**). |
| **SIGFPE** | Una operación aritmética errónea, tal como una división entre cero o una operación que provoca un desbordamiento de flujo. |
| **SIGILL** | Detección de una instrucción ilegal. |
| **SIGINT** | Recepción de una señal de atención interactiva. |
| **SIGSEGV** | Un acceso no permitido a almacenamiento. |
| **SIGTERM** | Una solicitud de terminación establecida en el programa. |

**Figura 14.7** Señales estándares de **signal.h**.

## 14.10 Manipulación de señales

Un *evento* inesperado, o *señal*, puede provocar que un programa termine prematuramente. Algunos eventos inesperados incluyen *interrupciones* (teclear ***<ctrl>-c*** en sistemas UNIX o Windows), *instrucciones ilegales*, *violaciones de segmentación*, *órdenes de finalización del sistema operativo* y *excepciones de punto flotante* (división entre cero, o multiplicación de valores de punto flotante demasiado grandes). La *biblioteca de manipulación de señales* (**signal.h**) proporciona la capacidad de *atrapar* eventos inesperados con la función **signal**. La función **signal** recibe dos argumentos, un entero para el número de señal y un apuntador a la función de manipulación de señales. Las señales pueden generarse por medio de la función **raise**, la cual toma como argumento un número entero como señal. La figura 14.7 resume las señales estándares definidas en el archivo de encabezado **signal.h**. La figura 14.8 muestra las funciones **signal** y **raise**.

```
/* Figura 14.8: fig14_08.c
   Uso de la manipulación de señales */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void manipSenal( int valorSenal ); /* prototipo */

int main()
{
   int i; /* contador utilizado para un ciclo de 10 repeticiones */
   int x; /* variable para almacenar valores aleatorios entre 1 y 50 */

   signal( SIGINT, manipSenal ); /* registra el manipulador de señal */
   srand( clock() );

   /* muestra los números de 1 a 100 */
   for ( i = 1; i <= 100; i++ ) {
      x = 1 + rand() % 50; /* genera números aleatorios hasta alcanzar SIGINT */

      /* alcanza SIGINT cuando x es 25 */
      if ( x == 25 ) {
         raise( SIGINT );
      } /* fin de if */

      printf( “%4d”, i );
```

**Figura 14.8** Manipulación de señales. (Parte 1 de 2.)

```

      /* muestra \n cuando i es un múltiplo de 10 */
      if ( i % 10 == 0 ) {
         printf( “\n” );
      } /* fin de if */

   } /* fin de for */

   return 0; /* indica terminación exitosa */

} /* fin de main */

/* manipula la señal */
void manipSenal( int valorSenal )
{
   int respuesta; /* respuesta del usuario a la señal (1 o 2) */

   printf( “%s%d%s\n%s”,
      “\nSenal de interrupcion ( “, valorSenal, ” ) recibida.”,
      “Desea continuar ( 1 = si o 2 = no )? ” );

   scanf( “%d”, &respuesta );

   /* verifica respuestas inválidas */
   while ( respuesta != 1 && respuesta != 2 ) {
      printf( “( 1 = si o 2 = no )? “ );
      scanf( “%d”, &respuesta );
   } /*fin de while */

   /* determina si es tiempo de terminar */
   if ( respuesta == 1 ) {

      /* registra el manipulador de señales para el siguiente SIGINT */
      signal( SIGINT, manipSenal );
   } /* fin de if */
   else {
      exit( EXIT_SUCCESS );
   } /* fin de else */

} /* fin de la función manipSenal */
```

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93
Senal de interrupcion ( 2 ) recibida.
Desea continuar ( 1 = si o 2 = no )? 1
  94  95  96
Senal de interrupcion ( 2 ) recibida.
Desea continuar ( 1 = si o 2 = no )? 2
```

**Figura 14.8** Manipulación de señales. (Parte 2 de 2.)

La figura 14.8 utiliza la función **signal** para atrapar una señal interactiva (**SIGINT**). La línea 15 llama a **signal** con **SIGINT** y un apuntador a la función **manipuladorSenal** (recuerde que el nombre de una función es un apuntador al principio de ésta). Cuando ocurre una señal de tipo **SIGINT**, el control pasa a la función **manipuladorSenal**, la cual imprime un mensaje y le da al usuario la opción de continuar la ejecución normal del programa. Si el usuario desea continuar la ejecución, reinicializa el manipulador de señales al llamar de nuevo a **signal**, con lo que el control regresa al punto en el programa en donde se detectó la señal. En este programa, la función **raise** (línea 24) se usa para simular una señal interactiva. Se elige un número aleatorio entre **1** y **50**. Si el número es **25**, se llama a **raise** para generar la señal. Por lo general, las señales interactivas se inicializan fuera del programa. Por ejemplo, digitar ***<ctrl>-c*** durante la ejecución del programa en un sistema UNIX o Windows genera una señal interactiva que termina la ejecución del programa. El manipulador de señales se puede utilizar para atrapar la señal interactiva que termina la ejecución del programa. El manipulador de señales puede utilizarse para atrapar la señal interactiva y prevenir que el programa termine.

## 14.11 Asignación dinámica de memoria: Las funciones calloc y realloc

En el capítulo 12, presentamos la noción de la asignación dinámica de memoria mediante el uso de la función **malloc**. Como establecimos en el capítulo 12, los arreglos son mejores que las listas ligadas para un ordenamiento rápido, la búsqueda y el acceso a datos. Sin embargo, por lo general los arreglos son *estructuras de datos estáticas*. La biblioteca general de utilidades (**stdlib.h**) proporciona otras dos funciones para asignación dinámica de memoria, **calloc** y **realloc**. Estas funciones pueden utilizarse para crear y modificar *arreglos dinámicos*. Como mostramos en el capítulo 7, es posible colocar un subíndice a un apuntador que apunta hacia un arreglo como si fuera un arreglo. Así, un apuntador a una porción contigua de memoria creada por **calloc** puede manipularse como un arreglo. La función **calloc** asigna memoria dinámicamente para un arreglo. El prototipo para **calloc** es

```
void *calloc( size_t nmemb, size_t tamanio );
```

Los dos argumentos representan el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**). La función **calloc** también inicializa en cero a los elementos del arreglo. La función devuelve un apuntador a la memoria asignada, o un apuntador **NULL** si la memoria no está asignada. La principal diferencia entre **malloc** y **calloc** es que **calloc** limpia la memoria que asigna y **malloc** no lo hace.

La función **realloc** modifica el tamaño de un objeto asignado por una llamada previa a **malloc**, **calloc** o **realloc**. El contenido original del objeto no se modifica si el monto de memoria asignada es mayor que el monto de memoria asignada previamente. De lo contrario, el contenido no se modifica hasta el tamaño del nuevo objeto. El prototipo de la función **realloc** es

```
void *realloc( void *ptr, size_t tamanio );
```

Los dos argumentos son: un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NULL**, **realloc** trabaja de forma idéntica a **malloc**. Si **tamanio** es **0** y **ptr** no es **NULL**, se libera la memoria del objeto. De lo contrario, si **ptr** no es **NULL** y **tamanio** es mayor que cero, **realloc** intenta asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no puede asignarse, el objeto al que apunta **ptr** no se modifica. La función **realloc** devuelve ya sea un apuntador a la reasignación de memoria, o un apuntador **NULL** para indicar que no se reasignó la memoria.

## 14.12 Saltos incondicionales con goto

A lo largo del libro hemos expresado la importancia de utilizar las técnicas de programación estructurada para construir software confiable que sea fácil de depurar, mantener y modificar. En algunos casos, el rendimiento es más importante que la estricta adherencia a las técnicas de la programación estructurada. En estos casos, es posible utilizar algunas técnicas de programación no estructurada. Por ejemplo, podemos utilizar **break** para terminar la ejecución de una estructura de repetición, antes de que la condición de continuación del ciclo se haga falsa. Esto ahorra repeticiones innecesarias del ciclo, si la tarea se completa antes de la terminación de éste.

```
1  /* Figura 14.9: fig14_09.c
2     Uso de goto */
3  #include <stdio.h>
4
5  int main()
6  {
7     int cuenta = 1; /* inicializa cuenta */
8
9     inicio: /* etiqueta */
10
11       if ( cuenta > 10 ) {
12          goto fin;
13       } /* fin de if */
14
15       printf( “%d  “, cuenta );
16       cuenta++;
17
18       goto inicio; /* ve a (goto) inicio en la línea 9 */
19
20    fin: /* etiqueta */
21       putchar( ‘\n’ );
22
23    return 0; /* indica terminación exitosa */
24
25 } /* fin de main */
```

```
1  2  3  4  5  6  7  8  9  10
```

**Figura 14.9** Instrucción **`goto`**.

Otro ejemplo de programación no estructurada es la *instrucción* **`goto`**, un salto incondicional. El resultado de la instrucción **`goto`** es un cambio en el flujo de control del programa a la primera línea después de la *etiqueta* especificada en la instrucción **`goto`**. Una etiqueta es un identificador seguido por dos puntos. Una etiqueta debe aparecer en la misma función que la instrucción **`goto`** que hace referencia a ella. La figura 14.9 utiliza instrucciones **`goto`** para realizar un ciclo de diez veces e imprimir, en cada ocasión, el valor del contador. Después de inicializar **`contador`** en **`1`**, la línea 11 prueba **`contador`** para determinar si es mayor que **`10`** (ignora la etiqueta inicio debido a que las etiquetas no realizan acción alguna). Si es así, el control se transfiere desde **`goto`** hasta la primera instrucción después de la etiqueta **`fin`** (la cual aparece en la línea 20). De lo contrario, las líneas 15 y 16 imprimen e incrementan **`contador`**, y el control se transfiere desde el **`goto`** (línea 18) a la primera instrucción después de la etiqueta **`inicio`** (la cual aparece en la línea 9).

En el capítulo 3, establecimos que solamente se requieren tres estructuras de control para escribir cualquier programa: secuencia, selección y repetición. Cuando se siguen las reglas de la programación estructurada, es posible crear estructuras de control profundamente anidadas, a partir de las cuales es difícil escapar de modo eficiente. Algunos programadores utilizan instrucciones **`goto`** en tales situaciones como una salida rápida de una estructura profundamente anidada. Esto elimina la necesidad de probar múltiples condiciones para escapar de una estructura de control.



### Tip de rendimiento 14.3

*La instrucción **`goto`** puede utilizarse para salir de modo eficiente de estructuras de control anidadas profundamente.*



### Observación de ingeniería de software 14.3

*La instrucción **`goto`** debe utilizarse solamente en aplicaciones orientadas al rendimiento. La instrucción **`goto`** no es estructurada y puede generar programas que sean más difíciles de depurar, mantener y modificar.*

## RESUMEN

- En muchos sistemas de cómputo es posible direccionar la entrada y la salida de un programa.
- La entrada se redirecciona desde la línea de comandos, usando el símbolo de redirección de entrada (**<**), o usando un símbolo de canalización (**|**).
- La salida desde la línea de comandos se redirecciona por medio del símbolo de redirección de salida (**>**), o del símbolo para agregar (**>>**). El símbolo de redirección de salida simplemente almacena la salida del programa en un archivo, y el símbolo para agregar adiciona la salida al final del archivo.
- Las macros y las definiciones del encabezado de argumentos variables **stdarg.h** proporcionan las capacidades necesarias para construir funciones con listas variables de argumentos.
- Los puntos suspensivos en el prototipo de una función indican un número variable de argumentos.
- El tipo **va_list** puede personalizarse para almacenar la información necesaria para las macros **va_start**, **va_arg** y **va_end**. Para acceder a los argumentos de una lista variable de argumentos, debe declararse un objeto de tipo **va_list**.
- La macro **va_start** se invoca antes de poder acceder a los argumentos de la lista variable de argumentos. La macro inicializa el objeto declarado con **va_list** para utilizarlo con las macros **va_arg** y **va_end**.
- La macro **va_arg** se desarrolla para formar una expresión con el valor y el tipo del siguiente argumento en la lista variable de argumentos. Cada invocación a **va_arg** modifica el objeto declarado con **va_list**, de modo que el objeto apunta al siguiente argumento de la lista.
- La macro **va_end** facilita un retorno normal desde una función a cuya lista variable de argumentos se hizo referencia mediante la macro **va_start**.
- En muchos sistemas es posible pasar argumentos a **main** desde la línea de comandos, al incluir los parámetros **int argc** y **char *argv[]** dentro de la lista de parámetros de **main**. El parámetro **argc** recibe el número de argumentos de la línea de comandos. El parámetro **argv** es un arreglo de cadenas en el que se almacena la lista de argumentos real de la línea de comandos.
- La definición de una función debe estar contenida en un solo archivo; no puede dividirse en dos o más archivos.
- Las variables globales deben declararse en cada archivo en donde se utilicen.
- Los prototipos de funciones pueden extender el alcance de una función más allá del archivo en el que se definen. Esto se lleva a cabo al incluir el prototipo de la función en cada archivo en donde se invoque a la función, y compilando juntos a los archivos.
- El especificador de clase de almacenamiento **static**, cuando se aplica a una variable global o a una función, evita que las utilice cualquier función que no esté definida dentro del mismo archivo. A esto se le llama vinculación interna. Las variables y las funciones globales que no son precedidas por **static** en sus definiciones tiene vinculación externa; se puede acceder a ellas desde otros archivos, si estos contienen las declaraciones apropiadas o los prototipos de las funciones.
- Por lo general, el especificador **static** se utiliza con las funciones de utilidad que son llamadas sólo por funciones dentro de un archivo en particular. Si no se requiere una función dentro de un archivo en particular, debe reforzarse el principio del menor privilegio mediante el uso de **static**.
- Cuando se construyen programas grandes en múltiples archivos fuente, la compilación del programa se hace tediosa, si al hacer los cambios pequeños se tiene que compilar todo el programa. Muchos sistemas proporcionan utilidades especiales que recompilan solamente el programa modificado. En los sistemas UNIX dicha utilidad se llama **make**. La utilidad **make** necesita un archivo llamado **makefile** que contiene instrucciones para compilar y enlazar el programa.
- La función **exit** fuerza al programa a terminar, como si se hubiera ejecutado normalmente.
- La función **atexit** registra a una función que debe invocarse cuando el programa termina de forma normal, es decir, cuando el programa termina al llegar al final de **main**, o cuando se invoca a **exit**.
- La función **atexit** toma como argumento un apuntador a una función. Las funciones que se invocan en la terminación del programa no pueden tener argumentos y no pueden devolver valor alguno. Se pueden registrar hasta 32 funciones para su ejecución durante la terminación del programa.
- La función **exit** toma un argumento. Por lo general, el argumento es la constante simbólica **EXIT_SUCCESS**, o la constante simbólica **EXIT_FAILURE**. Si se llama a **exit** con **EXIT_SUCCESS**, ésta devuelve el valor para la terminación exitosa, definido por la implementación, al ambiente de la función que hace la llamada. Si se llama a **exit** con **EXIT_FAILURE**, devuelve el valor de una terminación no exitosa, definido por la implementación.
- Cuando se invoca a la función **exit**, se invocan todas las funciones registradas en **atexit** en el orden inverso en el que se registraron, todos los flujos asociados con el programa se vacían y se cierran, y el control regresa al ambiente del anfitrión.

- El C estándar indica que cuando se utiliza **volatile** para calificar a un tipo, la naturaleza de acceso a un objeto de ese tipo depende de la implementación.
- C proporciona sufijos enteros y de punto flotante para especificar los tipos de constantes enteras y de punto flotante. Los sufijos enteros son: **u** o **U** para entero sin signo, **l** o **L** para entero largo, y **ul** o **UL** para un entero largo sin signo. Si no se coloca sufijo a una constante entera, el tipo se determina con el primer tipo capaz de almacenar un valor de dicho tamaño (primero **int**, después **long int**, después **unsigned long int**). Los sufijos de punto flotante son: **f** o **F** para **float**, y **l** o **L** para **long double**. Una constante de punto flotante que no tiene sufijo es de tipo **double**.
- C proporciona capacidades para procesar archivos binarios, pero algunos sistemas de cómputo no soportan archivos binarios. Si los archivos binarios no son soportados y se abre un archivo como binario, el archivo será procesado como un archivo de texto.
- La función **tmpfile** abre temporalmente un archivo en modo **"wb+"**. Aunque éste es un modo para archivo binario, algunos sistemas procesan archivos temporales como archivos de texto. Un archivo temporal existe hasta que se cierra con **fclose** o hasta que termina el programa.
- La biblioteca de manipulación de señales permite atrapar eventos inesperados con la función **signal**. Esta función recibe dos argumentos: un número entero de señal y un apuntador a la función de manipulación de señal.
- Las señales también pueden generarse con la función **raise** y un argumento entero.
- La biblioteca general de utilidades (**stdlib.h**) proporciona dos funciones para la asignación dinámica de memoria, **calloc** y **realloc**. Estas funciones pueden utilizarse para crear arreglos dinámicos.
- La función **calloc** recibe dos argumentos, el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**), e inicializa en cero a los elementos del arreglo. La función devuelve un apuntador a la memoria asignada, o un apuntador **NULL** si la memoria no está asignada.
- La función **realloc** modifica el tamaño de un objeto asignado por una llamada previa a **malloc**, **calloc** o **realloc**. El contenido original del objeto no se modifica, debido a que la cantidad de memoria asignada es mayor que la cantidad asignada previamente.
- La función **realloc** toma dos argumentos, un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NULL**, **realloc** funciona de modo idéntico a **malloc**. Si **tamanio** es **0** y el apuntador que recibe no es **NULL**, se libera la memoria para los objetos. De lo contrario, si **ptr** no es **NULL** y **tamanio** es mayor que cero, **realloc** intenta asignar un nuevo bloque de memoria para el objeto. Si no puede asignarse el nuevo espacio, el objeto al que apunta **ptr** permanece sin cambio. La función **realloc** devuelve un apuntador a la memoria reasignada, o un apuntador **NULL**.
- El resultado de la instrucción **goto** es un cambio en el flujo de control del programa. La ejecución del programa continúa en la primera instrucción después de la etiqueta especificada en la instrucción **goto**.
- Una etiqueta es un apuntador seguida por dos puntos. Una etiqueta debe aparecer en la misma función que la instrucción **goto** a la que hace referencia.

## *TERMINOLOGÍA*

archivo temporal
**argc**
argumentos de la línea de comandos
**argv**
arreglos dinámicos
**atexit**
atrapar
biblioteca de manipulación de señales
**calloc**
canalización
**const**
especificador de clase de almacenamiento **extern**
especificador de clase de almacenamiento **static**
evento
excepción de punto flotante
**exit**
**EXIT_FAILURE**
**EXIT_SUCCESS**
instrucción **goto**
instrucción ilegal
interrupción
lista de argumentos de longitud variable
**make**
**makefile**
**raise**
**realloc**
redirección de **E/S**
**signal**
símbolo de agregar a la salida **>>**
símbolo de canalización **(|)**
símbolo de redirección de entrada **(<)**
símbolo de redirección de salida **(>)**
sufijo de entero **unsigned** (**u** o **U**)
sufijo de entero **long** (**ul** o **UL**)
**stadarg.h**
sufijo de **long double** **(l** o **L)**
sufijo de **long int (l** o **L)**
sufijo de punto flotante **(f** o **F)**
**va_arg**
**va_end**
**va_list**
**va_start**
vinculación externa
vinculación interna
violación de segmentación
**volatile**
**signal.h**

## ERROR COMÚN DE PROGRAMACIÓN

**14.1** Colocar puntos suspensivos en medio de la lista de parámetros de una función, es un error de sintaxis. Los puntos suspensivos solamente pueden colocarse al final de la lista de parámetros.

## TIPS DE RENDIMIENTO

**14.1** Las variables globales incrementan el rendimiento debido a que se puede acceder a ellas directamente desde cualquier función, y se elimina la sobrecarga del paso de datos a funciones.

**14.2** Considere utilizar archivos binarios en lugar de archivos de texto, en aplicaciones que demandan alto rendimiento.

**14.3** La instrucción **`goto`** puede utilizarse para salir de modo eficiente de estructuras de control anidadas profundamente.

## TIP DE PORTABILIDAD

**14.1** Utilice archivos de texto, cuando escriba programas portables.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**14.1** Las variables globales deben evitarse, a menos que sean indispensables para el rendimiento de la aplicación, ya que éstas violan el principio del menor privilegio.

**14.2** Crear programas en distintos archivos fuente facilita la reutilización de software y la buena ingeniería de software. Las funciones pueden ser comunes a muchas aplicaciones. En dichas circunstancias esos archivos tienen que almacenarse en sus propios archivos fuente, y cada archivo fuente debe tener el archivo de encabezado correspondiente que contenga los prototipos de las funciones. Esto permite a los programadores de diferentes aplicaciones reutilizar el mismo código mediante la inclusión y compilación del archivo de encabezado apropiado para sus aplicaciones con el archivo fuente correspondiente.

**14.3** La instrucción **`goto`** debe utilizarse solamente en aplicaciones orientadas al rendimiento. La instrucción **`goto`** no es estructurada y puede generar programas que sean más difíciles de depurar, mantener y modificar.

## EJERCICIOS DE AUTOEVALUACIÓN

**14.1** Complete los espacios en blanco:

a) El símbolo ________ se utiliza para redireccionar la entrada de datos desde un archivo, en lugar de que sea desde el teclado.
b) El símbolo ________ se utiliza para redireccionar la salida de la pantalla para colocarla dentro de un archivo.
c) El símbolo ________ se utiliza para agregar la salida de un programa al final de un archivo.
d) Un símbolo ________ se utiliza para direccionar la salida de un programa para que sea la entrada de otro programa.
e) Un ________ en la lista de parámetros de una función indica que dicha función puede recibir un número variable de argumentos.
f) La macro ________ debe invocarse antes de poder acceder a los argumentos de una lista variable de argumentos.
g) La macro ________ se utiliza para acceder a los argumentos individuales de una lista variable de argumentos.
h) La macro ________ facilita un retorno normal desde una función a cuya lista variable de argumentos hace referencia la macro **`va_start`**.
i) El argumento ________ de **`main`** recibe el número de argumentos de la línea de comandos.
j) El argumento ________ de **`main`** almacena los argumentos de la línea de comandos como cadenas de caracteres.
k) La utilidad de UNIX ________ lee un archivo llamado ________ que contiene instrucciones para compilar y enlazar un programa que consta de múltiples archivos fuente. La utilidad solamente recompila un archivo si éste se modificó después de la última compilación.
l) La función ________ fuerza a un programa a terminar su ejecución.
m) La función ________ registra una función para que se invoque al término normal de un programa.

n) Un ____________ entero o de punto flotante puede agregarse a una constante entera o de punto flotante para especificar el tipo exacto de la constante.
o) La función ____________ abre un archivo temporal que existe hasta que se cierra o hasta que termina su ejecución.
p) La función ____________ puede utilizarse para atrapar eventos inesperados.
q) La función ____________ genera una señal desde adentro de un programa.
r) La función ____________ asigna memoria dinámicamente para cualquier arreglo, e inicializa los elementos en cero.
s) La función ____________ modifica el tamaño de un bloque de memoria previamente asignada de manera dinámica.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**14.1** a) De redirección de entrada (**<**). b) De redirección de salida (>). c) De agregar a la salida (>>). d) Canalización (**|**). e) Puntos suspensivos (**...**). f) **va_start**. g) **va_arg**. h) **va_end**. i) **argc**. j) **argv**. k) **make**, **makefile**. l) **exit**. m) **atexit**. n) Sufijo. o) **tmpfile**. p) **signal**. q) **raise**. r) **calloc**. s) **realloc**.

## EJERCICIOS

**14.2** Escriba un programa que calcule el producto de una serie de enteros que se pasen a la función **producto** por medio de una lista variable de argumentos. Pruebe su función con diversas llamadas, cada una con un número diferente de argumentos.

**14.3** Escriba un programa que imprima los argumentos de la línea de comandos del programa.

**14.4** Escriba un programa que ordene un arreglo de enteros en orden ascendente o descendente. El programa debe utilizar argumentos en la línea de comandos para pasar un argumento: **-a** para el orden ascendente, o **-d** para el orden descendente. [*Nota:* Éste es el formato estándar para pasar las opciones a un programa en UNIX.]

**14.5** Escriba un programa que coloque un espacio entre cada carácter en un archivo. El programa primero debe escribir el contenido del archivo a modificar dentro de un archivo temporal con espacios entre cada carácter, después, debe copiar el archivo de nuevo al archivo original. Esta operación debe sobrescribir los comentarios originales del archivo.

**14.6** Lea los manuales de su compilador para determinar qué señales son soportadas por la biblioteca de manipulación de señales (**signal.h**). Escriba un programa que contenga manipuladores de señales para las señales estándar **SIGABRT** y **SIGINT**. El programa debe verificar si atrapa estas señales llamando a la función **abort** para generar una señal de tipo **SIGABRT** y escribiendo ***<ctrl>c*** para generar una señal de tipo **SIGINT**.

**14.7** Escriba un programa que asigne de modo dinámico un arreglo de enteros. El tamaño del arreglo debe introducirse desde el teclado. Deben asignarse valores desde el teclado a los elementos del arreglo. Imprima los valores del arreglo. A continuación, reasigne la memoria para un arreglo con la mitad de elementos. Imprima los valores restantes en el arreglo para confirmar que coinciden con la primera mitad de los valores del arreglo original.

**14.8** Escriba un programa que tome nombres de archivos como dos argumentos en la línea de comandos, lea los caracteres del primer archivo, uno a la vez, y escriba los caracteres en orden inverso en el segundo archivo.

**14.9** Escriba un programa que utilice instrucciones **goto** para simular una estructura anidada que imprima un cuadrado de asteriscos de la siguiente manera.

```
*****
*   *
*   *
*   *
*****
```

El programa debe utilizar solamente las siguientes tres instrucciones **printf**:

```
printf( "*" );
printf( " " );
printf( "\n" );
```

# 15

# C++ como un "Mejor C"




## Objetivos

- Familiarizarse con las mejoras de C++, realizadas a C.
- Familiarizarse con la biblioteca estándar de C++.
- Comprender el concepto de las funciones **`inline`**.
- Crear y manipular referencias.
- Comprender el concepto de argumentos predeterminados.
- Comprender el rol que tiene el operador unario de resolución de alcance en el alcance en general.
- Sobrecargar funciones.
- Definir funciones que puedan realizar operaciones similares en diferentes tipos de datos.

*La forma siempre sigue a la función.*
Louis Henri Sullivan

*E pluribus unum.*
(*Uno compuesto por muchos.*)
Virgilio

*¡Oh!, que regrese el ayer, ruego al tiempo que vuelva.*
William Shakespeare

*Llámame Ismael.*
Herman Melville

*Cuando me llames así, sonríe.*
Owen Wister

## Plan general



## 15.1 Introducción

Ahora comenzamos con la segunda sección de este texto único. En los primeros catorce capítulos, presentamos un tratamiento completo sobre programación por procedimientos y sobre el diseño de programas de arriba hacia abajo en C. En la parte de este libro que corresponde a C++ (capítulos 15 a 23), presentamos tres paradigmas de programación adicionales: *la programación basada en objetos* (con clases, encapsulamiento, y sobrecarga de objetos y de operadores), *la programación orientada a objetos* (con herencia y polimorfismo) y la *programación genérica* (con plantillas de funciones y de clases), y enfatizaremos la creación de componentes reutilizables de software por medio de "la creación de clases valiosas". Una vez que estudiemos C++, presentaremos una introducción completa a la programación en Java (capítulos 24 a 30) utilizando las bibliotecas de clases para explorar la programación dirigida por eventos, la programación de gráficos, la programación de la interfaz gráfica de usuario (GUI) y la programación multimedia.

## 15.2 C++

C++ mejora muchas de las características de C y proporciona capacidades para *programación orientada a objetos* (*POO*) que representan una gran promesa para incrementar la productividad, calidad y reutilización del software. Este capítulo explica muchas de las mejoras de C++ realizadas a C.

Los diseñadores de C y los primeros que lo implementaron nunca anticiparon que el lenguaje se convertiría en un fenómeno (lo mismo se aplica para el sistema operativo UNIX). Cuando un lenguaje de programación se afianza tanto como C, los nuevos requerimientos demandan que el lenguaje evolucione, en lugar de que simplemente lo desplace un nuevo lenguaje. Bjarne Stroustrup desarrolló C++ en los laboratorios Bell, y originalmente lo llamó "C con clases". El nombre C++ incluye el operador de incremento de C (++), para indicar que C++ es una versión mejorada de C. C++ es un superconjunto de C, por lo que los programadores pueden utilizar un compilador de C++ para compilar programas de C existentes, y gradualmente evolucionar dichos programas a C++.

Los capítulos 15 a 23 proporcionan una introducción a la versión estandarizada de C++ en Estados Unidos a través de la *American National Standards Institute* (*ANSI*) y alrededor del mundo a través de la *International Standards Organization* (*ISO*). Nosotros hicimos un recorrido cuidadoso al documento estándar ANSI/ISO C++, y auditamos nuestra presentación contra éste para que estuviera completa y fuera adecuada. Sin embargo, C++ es un lenguaje rico, y existen ciertas sutilezas del lenguaje y temas avanzados que no cubrimos. Si us-

ted necesita detalles técnicos adicionales sobre C++, le sugerimos que lea el documento estándar de C++. Puede ordenar dicho documento desde el sitio Web de ANSI

**`http://www.ansi.org/`**

El título del documento es "Information Technology —Programming Languages— C++", y su número de documento es ISO/IEC 14882-1998. Si prefiere no comprar el documento, puede ver la versión antigua en borrador del estándar, en el sitio de la World Wide Web

**`http://www.cygnus.com/misc/wp/`**

Muchas características de la versión actual de C++ no son compatibles con implementaciones anteriores de C++, por lo que puede encontrar que algunos de los programas de este texto no funcionan en compiladores antiguos de C++.

## 15.3 Un programa sencillo: Suma de dos enteros

La figura 15.1 retoma el programa de suma de la figura 2.5 e ilustra muchas características importantes del lenguaje C++, así como algunas diferencias entre C y C++. [*Nota:* Los archivos en C tienen la extensión **`.c`** (minúscula). Los archivos en C++ pueden tener una variedad de extensiones: **`.cpp`**, **`.cxx`**, **`.C`** (mayúscula), etcétera. Nosotros utilizamos la extensión **`.cpp`**.]

Las líneas 1 y 2

```
//Figura 15.1: fig15_01.cpp
//Programa de suma
```

comienzan con **`//`**, las cuales indican que el resto de cada línea es un comentario. C++ le permite comenzar un comentario con **`//`** y utilizar el resto de la línea para comentar el texto. Los programadores en C++ también pueden utilizar comentarios al estilo C.

```
1  // Figura 15.1: fig15_01.cpp
2  // Programa de suma
3  #include <iostream>
4
5  int main()
6  {
7     int entero1;
8
9     std::cout << "Introduzca el primer entero\n";
10    std::cin >> entero1;
11
12    int entero2, suma;          // declaración
13
14    std::cout << "Introduzca el segundo entero\n";
15    std::cin >> entero2;
16    suma = entero1 + entero2;
17    std::cout << "La suma es " << suma << std::endl;
18
19    return 0;   // indica que el programa terminó de manera exitosa
20 } // fin de la función main
```

```
Introduzca el primer entero
45
Introduzca el segundo entero
72
La suma es 117
```

**Figura 15.1** Un programa de adición.

La directiva de preprocesador de C++ (línea 3)

```
#include <iostream>
```

exhibe el estilo del C++ estándar ANSI/ISO, para incluir archivos de encabezado de la biblioteca estándar. Esta línea le indica al preprocesador de C++ que incluya el contenido del *archivo de encabezado de flujo de entrada/salida* ***iostream***. Este archivo debe incluirse en cualquier programa que despliegue datos en la pantalla o que introduzca datos desde el teclado, utilizando el estilo de flujo de entrada/salida de C++. En el capítulo 21, explicaremos con detalle muchas características de **iostream**.

Al igual que en C, la línea 5 forma parte de todo programa en C++. La palabra reservada **int** que se encuentra a la izquierda de **main** indica que **main** "devuelve" un valor entero. Observe que en C, el programador no necesita especificar un tipo de valor de retorno para las funciones. Sin embargo, C++ sí requiere que el programador especifique un tipo de valor de retorno para todas las funciones, o el compilador generará un error.

### Error común de programación 15.1



*Omitir el tipo de valor de retorno en una definición de función de C++, es un error de sintaxis.*

La línea 7 es una *declaración de variable familiar*. Sin embargo, a diferencia de C, en donde las variables deben declararse en un bloque (es decir, un conjunto de llaves, **{}**) antes de cualquier instrucción ejecutable, las declaraciones de variables en C++ pueden colocarse casi en cualquier parte de un bloque. Esto se demuestra en la línea 12, en donde se declaran las variables **entero2** y **suma**.

### Buena práctica de programación 15.1

*Si prefiere colocar declaraciones al principio de una función, separe dichas declaraciones de las instrucciones ejecutables de esa función con una línea en blanco, para resaltar el lugar en donde terminan las declaraciones y en donde comienzan las instrucciones ejecutables.*

La instrucción de la línea 9 utiliza el *flujo de salida estándar* ***cout*** y el operador **<<** (*el operador de inserción de flujo*) para desplegar una cadena de texto. La salida y la entrada en C++ se logran por medio de *flujos* de caracteres. Por lo tanto, cuando la instrucción anterior se ejecuta, ésta envía el flujo de caracteres **Introduzca el primer entero** al *objeto de flujo de salida estándar*, ***std::cout***, que normalmente se "conecta" a la pantalla. Nos gusta pronunciar la instrucción anterior como "**cout** *obtiene* la cadena de caracteres **"Introduzca el primer entero\n"**".

La instrucción de la línea 10 utiliza el *objeto de flujo de entrada* ***cin*** y el *operador de extracción de flujo*, **>>**, para obtener un valor desde el teclado. Utilizar el operador de extracción de flujo hace que **std::cin** tome la entrada de caracteres del flujo de entrada estándar, el cual normalmente es el teclado. A nosotros nos gusta pronunciar la instrucción anterior como, "**std::cin** *da* un valor a **entero1**", o simplemente "**std::cin** *da* **entero1**".

Cuando la computadora ejecuta la instrucción anterior, ésta espera que el usuario introduzca un valor para la variable **entero1**. El usuario responde escribiendo un entero (como caracteres), y después oprimiendo la tecla *Entrar*. Posteriormente la computadora convierte la representación del carácter del número en un entero y asigna este valor a la variable **entero1**.

La línea 14 imprime en la pantalla las palabras **Introduzca el segundo entero**, y después se posiciona en el comienzo de la siguiente línea. Esta instrucción indica al usuario que haga algo. La línea 15 obtiene del usuario un valor para la variable **entero2**.

La línea 17 despliega la cadena de caracteres **La suma es**, seguida por el valor numérico de la variable **suma**, seguido por **std::endl** (**endl** es una abreviatura para "final de línea"), también conocido como *manipulador de flujo*. El manipulador **std::endl** despliega una nueva línea, después "desaloja el buffer de salida". Esto simplemente significa que, en algunos sistemas en los que las salidas se acumulan en la máquina hasta que "vale la pena desplegarlas en la pantalla", **std::endl** ocasiona que cualquier salida acumulada se despliegue en ese momento.

Observe que colocamos **std::** antes de **cout**, **cin** y **endl**. Esto es necesario cuando utilizamos la directiva de preprocesador **#include <iostream>**. La notación **std::cout** especifica que estamos utilizando un nombre, en este caso **cout**, que pertenece al "espacio de nombres" **std**. Los espacios de nombres son una característica avanzada de C++ que no explicamos en estos capítulos introductorios de C++. Por ahora, simplemente recuerde incluir **std::** antes de cada mención de **cout**, **cin** y **cerr** en un programa. Esto

puede ser engorroso; en la figura 15.3 presentamos la instrucción **using**, la cual nos permite evitar **std::** antes de cada utilización de un espacio de nombre **std**.

Vea que la instrucción de la línea 17 despliega diversos valores de diferentes tipos (por ejemplo, cadenas, **double**, **entero**s, etcétera). El operador de inserción de flujo "sabe" cómo desplegar cada pieza de datos. Utilizar varios operadores de inserción de flujo (**<<**) en una sola instrucción se conoce como *operaciones de inserción de flujo para concatenación, encadenamiento o en cascada*. Por lo tanto, no es necesario tener diversas instrucciones de salida para desplegar varias piezas de datos. Vea el apéndice C, para que obtenga una lista completa de los operadores de C++.

Los cálculos también pueden realizarse en instrucciones de salida. Nosotros pudimos haber combinado las instrucciones de las líneas 16 y 17 en la instrucción

```
std::cout << "La suma es " << entero1 + entero2 << std::endl;
```

con lo que eliminaríamos la necesidad de la variable **suma**.

Una poderosa característica de C++ es que los usuarios pueden crear sus propios tipos de datos. Ellos pueden entonces "enseñar" a C++ cómo introducir y desplegar valores de estos nuevos tipos de datos por medio de los operadores **>>** y **<<** (a esto se le conoce como *sobrecarga de operadores*; un tema que abordaremos en el capítulo 18).

## 15.4 Biblioteca estándar de C++

Los programas en C++ se construyen con dos bloques de construcción principales, *funciones* y tipos de datos definidos por el usuario llamados *clases*, los cuales analizaremos con detalle en el siguiente capítulo. La mayoría de los programadores en C++ aprovechan las ricas colecciones de clases y funciones existentes en la biblioteca estándar de C++. Por lo tanto, en realidad hay dos partes por aprender en el "mundo" de C++. La primera es aprender el lenguaje mismo C++ y la segunda es aprender cómo utilizar las clases y las funciones de la biblioteca estándar de C++. A lo largo del libro, explicaremos muchas de estas clases y funciones. Los fabricantes de compiladores generalmente proporcionan las bibliotecas de clases. Los fabricantes independientes de software proporcionan muchas de las bibliotecas de clases de propósitos especiales.



**Observación de ingeniería de software 15.1**

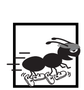
*Utilice un "método de construcción en bloques" para crear programas. Evite reinventar la rueda. Utilice piezas existentes en donde sea posible; a esto se le llama "reutilización de software", y es básica para la programación orientada a objetos.*



**Observación de ingeniería de software 15.2**

*Cuando programe en C++, normalmente utilice los siguientes bloques de construcción: clases y funciones de la biblioteca estándar de C++, clases y funciones que genere usted mismo, y clases y funciones de otras bibliotecas populares provistas por fabricantes de terceros.*

La ventaja de crear sus propias funciones y clases es que sabrá exactamente cómo funcionan. Usted podrá examinar el código en C++. La desventaja es el consumo de tiempo y el complejo esfuerzo que implica el diseñar, desarrollar y mantener nuevas funciones y clases que sean correctas y que operen de manera eficiente.



**Tip de rendimiento 15.1**

*Utilizar las funciones y clases de la biblioteca estándar en lugar de escribir las suyas, puede mejorar el rendimiento del programa, ya que este software está cuidadosamente escrito para que funcione correcta y eficientemente.*



**Tip de portabilidad 15.1**

*Utilizar las funciones y clases de la biblioteca estándar en lugar de escribir las suyas, puede mejorar la portabilidad del programa, ya que este software está incluido en casi todas las implementaciones de C++.*

## 15.5 Archivos de encabezados

Cada biblioteca estándar tiene un *archivo de encabezado* correspondiente que contiene los prototipos de función para todas las funciones de esa biblioteca, y las definiciones de varios tipos de datos y constantes necesarias para esas funciones.



### Observación de ingeniería de software 15.3

*En C++ son necesarios los prototipos de función. Utilice las directivas de preprocesador* **`#include`** *para obtener los prototipos de función de la biblioteca estándar. También utilice* **`#include`** *para obtener los archivos de encabezado que contienen los prototipos de función utilizados por usted y/o por los miembros de su grupo.*

La figura 15.2 lista algunos archivos de encabezado de la biblioteca estándar de C++ que pueden incluirse en programas de C++. Los archivos de encabezado que terminan en **`.h`** son archivos de encabezado con el “viejo estilo”, los cuales han sido reemplazados por los archivos de encabezado de la biblioteca estándar de C++.

El programador puede crear archivos de encabezado personalizados. Los archivos de encabezado definidos por el programador deben finalizar con **`.h`**. Un archivo de encabezado definido por el programador puede incluirse por medio de la directiva de preprocesador **`#include`**. Por ejemplo, el archivo de encabezado **`cuadrado.h`** puede incluirse en nuestro programa, colocando la directiva

```
#include “cuadrado.h”
```

al principio del programa.

| Archivo de encabezado de la biblioteca estándar | Explicación |
|---|---|
| **`<cassert>`** | Contiene macros e información para agregar diagnósticos que ayuden en la depuración de programas. La antigua versión de este archivo de encabezado es **`<assert.h>`**. |
| **`<cctype>`** | Contiene prototipos de función para funciones que evalúan ciertas propiedades de los caracteres, las cuales pueden utilizarse para convertir letras minúsculas a mayúsculas y viceversa. Este archivo de encabezado reemplaza a **`<ctype.h>`**. |
| **`<cfloat>`** | Contiene los límites del sistema con respecto al tamaño de los números de punto flotante. Este archivo de encabezado reemplaza a **`<float.h>`**. |
| **`<climits>`** | Contiene los límites del sistema para números enteros. Este archivo de encabezado reemplaza a **`<limits.h>`**. |
| **`<cmath>`** | Contiene prototipos de función de la biblioteca de funciones matemáticas. Este archivo de encabezado reemplaza a **`<math.h>`**. |
| **`<cstdio>`** | Contiene prototipos de función para las funciones de entrada/salida de la biblioteca estándar, e información que éstas utilizan. Este archivo de encabezado reemplaza a **`<stdio.h>`**. |
| **`<cstdlib>`** | Contiene prototipos de función para la conversión de números a texto, de texto a número, para asignación de memoria, para números aleatorios y otras funciones útiles. Este archivo de encabezado reemplaza a **`<stdlib.h>`**. |
| **`<cstring>`** | Contiene prototipos de función para funciones de procesamiento de cadenas al estilo C. Este archivo de encabezado reemplaza a **`<string.h>`**. |
| **`<ctime>`** | Contiene prototipos de función y tipos para manipular fechas y horas. Este archivo de encabezado reemplaza a **`<time.h>`**. |
| **`<iostream>`** | Contiene prototipos de función para las funciones de entrada y salida estándar. Este archivo de encabezado reemplaza a **`<iostream.h>`**. |
| **`<iomanip>`** | Contiene prototipos de función para los manipuladores de flujo que permiten dar formato a los flujos de datos. Este archivo de encabezado reemplaza a **`<iomanip.h>`**. |
| **`<fstream>`** | Contiene prototipos para funciones que realizan entradas desde archivos en disco, y salidas hacia archivos en disco. Este archivo de encabezado remplaza a **`<fstream.h>`**. |
| **`<utility>`** | Contiene clases y funciones que son utilizadas por muchos de los archivos de encabezado de la biblioteca estándar. |

**Figura 15.2** Archivos de encabezado de la biblioteca estándar. (Parte 1 de 2.)

| Archivo de encabezado de la biblioteca estándar | Explicación |
|---|---|
| **`<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>`** | Estos archivos de encabezado contienen clases que implementan los contenedores de la biblioteca estándar. Los contenedores se utilizan para almacenar datos durante la ejecución de un programa. |
| **`<functional>`** | Contiene clases y funciones utilizadas por los algoritmos de la biblioteca estándar. |
| **`<memory>`** | Contiene clases y funciones utilizadas por la biblioteca estándar para asignar memoria a los contenedores de la biblioteca estándar. |
| **`<iterator>`** | Contiene clases para el acceso a datos en los contenedores de la biblioteca estándar. |
| **`<algorithm>`** | Contiene funciones para manipular datos en los contenedores de la biblioteca estándar. |
| **`<exception>, <stdexcept>`** | Estos archivos de encabezado contiene clases que se utilizan para el manejo de excepciones (las cuales explicamos en el capítulo 23). |
| **`<string>`** | Contiene la definición de la clase **`string`** de la biblioteca estándar. |
| **`<sstream>`** | Contiene los prototipos para las funciones que realizan entradas desde cadenas en memoria, y salidas hacia cadenas en memoria. |
| **`<locale>`** | Contiene clases y funciones normalmente utilizadas en el procesamiento de flujo, para procesar datos en la forma natural de diferentes lenguajes (por ejemplo, en formatos de dinero, ordenamiento de cadenas, presentación de caracteres, etcétera). |
| **`<limits>`** | Contiene clases para definir los límites de tipos de datos numéricos en cada plataforma de cómputo. |
| **`<typeinfo>`** | Contiene clases para identificación de tipos en tiempo de ejecución (es decir, se determina el tipo de los datos en tiempo de ejecución). |

**Figura 15.2** Archivos de encabezado de la biblioteca estándar. (Parte 2 de 2.)

## 15.6 Funciones `inline`

Como vimos en el capítulo 5, implementar un programa en C como un conjunto de funciones es bueno desde un punto de vista de ingeniería de software, pero las llamadas a función involucran una sobrecarga en tiempo de ejecución. C++ proporciona *funciones* ***`inline`*** para ayudar a reducir la sobrecarga de llamadas a funciones; en especial a pequeñas funciones. Cuando en la definición de una función, antes del tipo de retorno de la función, se coloca el calificador **`inline`**, éste “aconseja” al compilador que genere una copia del código de la función, para evitar una llamada a función. La desventaja es que se insertan muchas copias del código de la función en el programa (lo que hace que el programa sea más largo), en lugar de una sola copia de la función a la que se le pasa el control cada vez que se llama a dicha función. El compilador puede ignorar el calificador **`inline`**, y generalmente lo hace en todos los casos, excepto en el de las funciones más pequeñas.



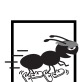
**Observación de ingeniería de software 15.4**

*Cualquier modificación a una función* ***`inline`*** *podría requerir que todos los clientes de dicha función se recompilaran. Esto puede ser importante en algunas situaciones de desarrollo de programas y de mantenimiento.*



**Buena práctica de programación 15.2**

*El calificador* ***`inline`*** *debe utilizarse sólo con pequeñas funciones que se utilicen con frecuencia.*



**Tip de rendimiento 15.2**

*Utilizar funciones* ***`inline`*** *puede reducir el tiempo de ejecución, pero incrementar el tamaño del programa.*

La figura 15.3 utiliza la función **`inline cubo`** para calcular el volumen de un cubo. Las líneas 6 a 8

```
using std::cout;
using std::cin;
using std::endl;
```

utilizan *instrucciones* **`using`** para ayudarnos a eliminar la necesidad de repetir el prefijo **`std::`**. Una vez que incluimos estas instrucciones **`using`**, podemos escribir **`cout`**, en lugar de **`std::cout`**, **`cin`** en lugar de **`std::cin`**, y **`endl`** en lugar de **`std::endl`**, en el resto del programa. [*Nota:* A partir de este momento, cada ejemplo de C++ contendrá una o más instrucciones **`using`**.]

```
1  // Figura 15.3: fig15_03.cpp
2  // Uso de una función inline para calcular
3  // el volumen de un cubo.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 inline double cubo( const double s ) { return s * s * s; }
11
12 int main()
13 {
14   double lado;
15
16   for ( int k = 1; k < 4; k++ ) {
17       cout << “Introduzca la longitud de un lado del cubo:  “;
18       cin >> lado;
19       cout << “El Volumen del cubo con lado “
20            << lado << “ es “ << cubo( lado ) << endl;
21   } // fin de for
22
23    return 0;
24 } // fin de la función main
```

```
Introduzca la longitud de un lado del cubo:  1.0
El Volumen del cubo con lado 1 es 1
Introduzca la longitud de un lado del cubo:  2.3
El Volumen del cubo con lado 2.3 es 12.167
Introduzca la longitud de un lado del cubo:  5.4
El Volumen del cubo con lado 5.4 es 157.464
```

**Figura 15.3** Uso de una función **`inline`** para calcular el volumen de un cubo.

Observe la declaración de la variable **`k`** en el ciclo **`for`** (línea 16). C++ da al programador la opción de declarar una variable de control para el ciclo **`for`**, en la sección de inicialización del encabezado **`for`**. Las variables de control declaradas en el encabezado **`for`** pueden utilizarse sólo en el cuerpo de la estructura **`for`**, mientras el valor de la variable de control sea desconocida fuera del encabezado y cuerpo de la estructura del **`for`**. Todas las demás estructuras de control de C++ son las mismas que en C.

La condición de la línea 16

```
k < 4
```

da como resultado un valor **`0`** (falso) o uno diferente de cero (verdadero). Esto es consistente con C. Sin embargo, C++ agrega un tipo de dato ***`bool`*** para representar un valor booleano. Una variable **`bool`** puede asignarse a un valor entero, es decir, a la palabra reservada ***`true`*** o la palabra reservada ***`false`***. Comenzaremos a utilizar las palabras reservadas **`bool`**, **`true`** y **`false`**, en los siguientes capítulos de C++. La figura 15.4 lista las palabras reservadas comunes de C y C++, y las palabras reservadas exclusivas de C++.

**Palabras reservadas de C++**

*Palabras reservadas comunes de los lenguajes de programación C y C++*

| | | | | |
|---|---|---|---|---|
| `auto` | `break` | `case` | `char` | `const` |
| `continue` | `default` | `do` | `double` | `else` |
| `enum` | `extern` | `float` | `for` | `goto` |
| `if` | `int` | `long` | `register` | `return` |
| `short` | `signed` | `sizeof` | `static` | `struct` |
| `switch` | `typedef` | `union` | `unsigned` | `void` |
| `volatile` | `while` | | | |

*Palabras reservadas exclusivas de C++*

| | | | | |
|---|---|---|---|---|
| `asm` | `bool` | `catch` | `class` | `const_cast` |
| `delete` | `dynamic_cast` | `explicit` | `false` | `friend` |
| `inline` | `mutable` | `namespace` | `new` | `operator` |
| `private` | `protected` | `public` | `reinterpret_cast` | |
| `static_cast` | `template` | `this` | `throw` | `true` |
| `try` | `typeid` | `typename` | `using` | `virtual` |
| `wchar_t` | | | | |

**Figura 15.4** Palabras reservadas de C++.

## 15.7 Referencias y parámetros de referencias

En muchos lenguajes de programación hay dos formas de invocar funciones: *las llamadas por valor* y las *llamadas por referencia*. Cuando se pasa un argumento por medio de una llamada por valor, se hace una *copia* del valor del argumento y se pasa a la función invocada. Modificar la copia no afecta el valor original de la variable en la función que hace la llamada. Esto evita efectos colaterales accidentales que entorpecen grandemente el desarrollo correcto y confiable de los sistemas de software. Cada uno de los argumentos que fueron pasados en los programas anteriores de este capítulo, fueron pasados mediante llamadas por valor.



**Tip de rendimiento 15.3**

*Una desventaja de las llamadas por valor es que, si se está pasando un elemento de datos grande, copiar dicho elemento puede implicar demasiado tiempo de ejecución.*

En esta sección presentamos los *parámetros por referencias*; la segunda técnica que C++ proporciona para realizar llamadas por referencia. En el capítulo 7 presentamos la primera técnica: apuntadores. Con una llamada por referencia, la función que realiza la llamada da a la función llamada la capacidad de acceder directamente a los datos de la función que la llama, y de modificar dichos datos si así lo decide.



**Tip de rendimiento 15.4**

*Una llamada por referencia es buena por motivos de rendimiento, ya que ésta elimina la sobrecarga de copiar grandes cantidades de datos.*



**Observación de ingeniería de software 15.5**

*Una llamada por referencia puede debilitar la seguridad, ya que la función llamada puede corromper los datos de la función que la llamó.*

Un parámetro por referencia es un alias de su argumento correspondiente. Para indicar que un parámetro de función pasa por referencia, simplemente coloque un amperson (**&**) después del tipo del parámetro en el prototipo de la función; utilice la misma convención cuando liste el tipo del parámetro en el encabezado de función. Por ejemplo, la declaración

```
int &cuenta
```

en un encabezado de función puede leerse como “**`cuenta`** es una referencia a un **`int`**”. En la llamada de función, simplemente mencione a la variable por su nombre, y ésta será pasada por referencia. Mencionar a la variable por medio del nombre de su parámetro en el cuerpo de la función llamada, en realidad hace referencia a la variable original de la función que hace la llamada, y la función llamada puede modificar directamente a la variable original. Como siempre, el prototipo y el encabezado de la función deben coincidir.

La figura 15.5 compara la llamada por valor y la llamada por referencia mediante parámetros por referencia. Los “estilos” de los argumentos en las llamadas a **`cuadradoPorValor`** y **`cuadradoPorReferencia`** son idénticos, mientras ambas variables se mencionen simplemente por sus nombres. Sin verificar los prototipos de las funciones o las definiciones de éstas, no es posible decir, a partir de las llamadas, si cualquiera de las funciones puede modificar sus argumentos. Sin embargo, los prototipos de las funciones son obligatorios, por lo que el compilador no tiene problemas para resolver la ambigüedad.

```
// Figura 15.05: fig15_05.cpp
// Comparación de una llamada por valor y una llama por referencia
// mediante referencias.
#include <iostream>

using std::cout;
using std::endl;

int cuadradoPorValor( int );
void cuadradoPorReferencia( int & );

int main()
{
   int x = 2, z = 4;

   cout << “x = “ << x << “ antes de cuadradoPorValor\n”
        << “Valor devuelto por cuadradoPorValor: “
        << cuadradoPorValor( x ) << endl
        << “x = “ << x << “ despues de cuadradoPorValor\n” << endl;

   cout << “z = “ << z << “ despues de cuadradoPorReferencia” << endl;
   cuadradoPorReferencia( z );
   cout << “z = “ << z << “ despues de cuadradoPorReferencia” << endl;

   return 0;
} // fin de la función main

int cuadradoPorValor( int a )
{
   return a *= a;   // argumento de la llamada no modificada
} // fin de la función cuadradoPorValor

void cuadradoPorReferencia( int &cRef )
{
   cRef *= cRef;    // argumento de la llamada modificada
} // fin de la función cuadradoPorReferencia
```

```
x = 2 antes de cuadradoPorValor
Valor devuelto por cuadradoPorValor: 4
x = 2 despues de cuadradoPorValor

z = 4 despues de cuadradoPorReferencia
z = 16 despues de cuadradoPorReferencia
```

**Figura 15.5** Un ejemplo de una llamada por referencia.



### Error común de programación 15.2

*Los parámetros por referencia se mencionan sólo por nombre en el cuerpo de la función llamada, por lo que el programador podría tratar inadvertidamente a los parámetros por referencia como parámetros de una llamada por valor. Esto puede ocasionar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función que hace la llamada.*

### Tip de rendimiento 15.5

*Para pasar objetos grandes, utilice un parámetro por referencia constante para simular la apariencia y seguridad de una llamada por valor y para evitar la sobrecarga de pasar una copia de ese gran objeto.*

Observe que colocamos un **&** en la lista de parámetros de la función **cuadradoPorReferencia**. Algunos programadores en C++ prefieren escribir **int& cRef**, en lugar de **int &cRef**.



### Observación de ingeniería de software 15.6

*Por razones combinadas de claridad y rendimiento, muchos programadores en C++ prefieren que los argumentos modificables sean pasados por medio de apuntadores, que los argumentos pequeños no modificables pasen por medio de una llamada por valor, y que los argumentos grandes no modificables pasen por medio de referencias a constantes.*

Las referencias también pueden utilizarse como alias de otras variables dentro de una función. Por ejemplo, el código

```
int cuenta = 1;       // declara la variable entera cuenta
int &cRef = cuenta;   // crea cRef como un alias de cuenta
++cRef;               // incrementa cuenta (por medio de su alias)
```

incrementa la variable **cuenta** por medio de su alias **cRef**. Las variables de referencia deben inicializarse en sus declaraciones (vea las figuras 15.6 y 15.7), y no pueden reasignarse como alias de otras variables. Una vez que se declara una referencia como un alias de otra variable, todas las operaciones supuestamente realizadas sobre el alias (es decir, sobre la referencia) en realidad se realizan sobre la variable original misma. El alias es tan solo otro nombre para la variable original. Ni tomar la dirección de una referencia, ni comparar referencias, ocasiona errores de sintaxis; en cambio, cada operación realmente ocurre sobre la variable para la cual, la re-

```
// Figura 15.6: fig15_06.cpp
// Las referencias se deben inicializar
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int x = 3, &y = x;  // y no es un alias para x

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
} // fin de la función main
```

```
x = 3
y = 3
x = 7
y = 7
```

**Figura 15.6** Uso de una referencia inicializada.

```
// Figura 15.7: fig15_07.cpp
// Las referencias se deben inicializar
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int x = 3, &y;   // Error: y se debe inicializar

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
} // fin de la función main
```

*Mensaje de error del compilador Microsoft Visual C++*

```
fig15_07.cpp(10) : error C2530: 'y' : references must be initialized
```

**Figura 15.7** Intento de utilizar una referencia no inicializada.

ferencia es su alias. Un argumento de referencia debe ser un *lvalue*, no una constante o expresión que devuelva un *rvalue*.



**Error común de programación 15.3**

*No inicializar una variable de referencia cuando ésta se declara, es un error de sintaxis.*



**Error común de programación 15.4**

*Intentar reasignar una referencia previamente declarada para que sea un alias de otra variable, es un error lógico. El valor de la otra variable simplemente se asigna a la ubicación para la cual, la referencia ya es un alias.*



**Error común de programación 15.5**

*Declarar varias referencias en una instrucción, mientras se asume que el* **`&`** *se distribuye a lo largo de una lista de nombres de variables separados por comas. Para declarar las variables* **`x`**, **`y`** *y* **`z`** *como referencias a enteros, utilice la notación* **`int &x = a, &y = b, &z = c;`** *en lugar de utilizar la notación incorrecta* **`int& x = a, y = b, z = c;`** *o la otra notación común incorrecta* **`int &x, y, z;`**.

Las funciones pueden devolver referencias, pero esto puede ser peligroso. Cuando se devuelve una referencia a una variable declarada en la función llamada, la variable debe declararse como **`static`** dentro de esa función. De lo contrario, la referencia se referirá a una variable automática que se descarta cuando la función termina; se dice que tales variables son "indefinidas", y el comportamiento del programa sería impredecible (algunos compiladores despliegan advertencias cuando esto se hace). Las referencias a variables indefinidas se conocen como *referencias indefinidas*.



**Error común de programación 15.6**

*Devolver un apuntador o referencia a una variable automática en una función llamada, es un error lógico. Algunos compiladores despliegan una advertencia, cuando esto ocurre en un programa.*

## 15.8 Argumentos predeterminados y listas de parámetros vacías

Las llamadas a funciones comúnmente pasan un valor particular de un argumento. El programador puede especificar dicho argumento como un *argumento predeterminado*, y puede proporcionar un valor predeterminado

para ese argumento. Cuando un argumento predeterminado se omite en una llamada a función, el compilador inserta el valor predeterminado de dicho argumento y lo pasa en la llamada.

Los argumentos predeterminados deben ser los que se encuentren más a la derecha de la lista de parámetros de la función. Cuando uno llama a una función con dos o más argumentos predeterminados, si uno de los argumentos que se omite no es el que se encuentra más a la derecha de la lista de argumentos, entonces todos los argumentos a la derecha de ese argumento también deben omitirse. Los argumentos predeterminados deben especificarse con la primera ocurrencia del nombre de la función; por lo general, en el prototipo. Los valores predeterminados pueden ser constantes, variables globales o llamadas a función.

La figura 15.8 muestra el uso de argumentos predeterminados para calcular el volumen de una caja. El prototipo de función para **volumenCaja** en la línea 8 especifica que a los tres argumentos se les proporcionaron valores predeterminados de **1**. Observe que los valores predeterminados deben definirse sólo en el prototipo de función. También observe que para efectos de legibilidad proporcionamos nombres de variables en el prototipo de la función. Como siempre, los nombres de variables no son necesarios en los prototipos de función.

```
1  // Figura 15.8: fig15_08.cpp
2  // Uso de argumentos predeterminados
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int volumenCaja( int longitud = 1, int ancho = 1, int alto = 1 );
9
10 int main()
11 {
12    cout << “El volumen predeterminado de la caja es : “ << volumenCaja()
13         << “\n\nEl volumen de una caja de longitud 10,\n”
14         << “ancho 1 y altura 1 es: “ << volumenCaja( 10 )
15         << “\n\nEl volumen de una caja de longitud 10,\n”
16         << “ancho 5 y altura 1 es: “ << volumenCaja( 10, 5 )
17         << “\n\nEl volumen de una caja de longitud 10,\n”
18         << “ancho 5 y altura 2 es: “ << volumenCaja( 10, 5, 2 )
19         << endl;
20
21    return 0;
22 } // fin de la función main
23
24 // Calcula el volumen de la caja
25 int volumenCaja( int longitud, int ancho, int altura )
26 {
27    return longitud * ancho * altura;
28 } // fin de la función volumenCaja
```

```
El volumen predeterminado de la caja es : 1

El volumen de una caja de longitud 10,
ancho 1 y altura 1 es: 10

El volumen de una caja de longitud 10,
ancho 5 y altura 1 es: 50

El volumen de una caja de longitud 10,
ancho 5 y altura 2 es: 100
```

**Figura 15.8** Uso de argumentos predeterminados.

La primera llamada a **`volumenCaja`** (línea 12) no especifica argumentos y, por lo tanto, utiliza tres valores predeterminados. La segunda llamada (línea 14) pasa un argumento **`longitud`**, y luego utiliza los valores predeterminados para los argumentos **`ancho`** y **`alto`**. La tercera llamada (línea 16) pasa argumentos para **`longitud`** y **`ancho`**, y luego utiliza el valor predeterminado para el argumento **`alto`**. La última llamada (línea 18) pasa argumentos para **`longitud`**, **`ancho`** y **`alto`**, y luego ya no utiliza valores predeterminados.



**Buena práctica de programación 15.3**

*Utilizar argumentos predeterminados simplifica la escritura de llamadas a función. Sin embargo, algunos programadores sienten que especificar explícitamente todos los argumentos es más claro.*



**Error común de programación 15.7**

*Especificar e intentar utilizar un argumento predeterminado que no sea el que se encuentra más a la derecha (mientras a los demás argumentos no se les asigne simultáneamente valores predeterminados) es un error de sintaxis.*

C++, como C, permite al programador dejar vacía la lista de parámetros de una función. En C++, una lista de parámetros vacía se especifica escribiendo **`void`**, o nada, en los paréntesis. Los prototipos

```
void imprime1();
void imprime2( void );
```

especifican que las funciones **`imprime1`** e **`imprime2`** no toman argumentos y no devuelven valores. A diferencia de los nombres de función, estos prototipos son equivalentes.



**Tip de portabilidad 15.2**

*En C++, el significado de una lista de parámetros vacía es dramáticamente diferente que en C. En C, significa que toda verificación de argumentos está deshabilitada (es decir, la llamada de función puede pasar cualquier argumento que desee). En C++, significa que la función no toma argumentos. Por lo tanto, los programas en C que utilizan estas características podrían reportar errores de sintaxis cuando se compilan en C++.*

## 15.9 Operador unario de resolución de alcance

Recuerde de nuestra explicación sobre las reglas de alcance que vimos en el capítulo 5, que es posible declarar variables locales y globales con el mismo nombre. Esto ocasiona que la variable global esté "oculta" para la variable local, en un alcance local. C++ proporciona el *operador unario de resolución de alcance* (**::**) para proporcionar acceso a una variable global cuando ésta ha sido oculta para una variable local con el mismo nombre, en un alcance local. Sin embargo, el operador unario de resolución de alcance no puede utilizarse para acceder a una variable local del mismo nombre en un bloque externo. Tal y como en C, se puede acceder directamente a una variable global sin el operador unario de resolución de alcance, si el nombre de la variable global no es el mismo que el de la variable local en alcance.

La figura 15.9 muestra al operador unario de resolución de alcance con una variable global y una local con el mismo nombre. Para enfatizar que las versiones global y local de una variable **`PI`** son diferentes, el programa declara a una de las variables como **`double`** y a la otra como **`float`**.

```
// Figura 15.9: fig15_09.cpp
// Uso del operador unario de resolución de alcance
#include <iostream>

using std::cout;
using std::endl;
using std::ios;

#include <iomanip>

using std::setprecision;
```

**Figura 15.9** Uso del operador unario de resolución de alcance. (Parte 1 de 2.)

```
12 using std::setiosflags;
13 using std::setw;
14
15 const double PI = 3.14159265358979;
16
17 int main()
18 {
19    const float PI = static_cast< float >( ::PI );
20
21    cout << setprecision( 20 )
22         << "  Valor local float de PI = " << PI
23         << "\nValor global double de PI = " << ::PI << endl;
24
25    cout << setw( 28 ) << "Valor float local de PI = "
26         << setiosflags( ios::fixed | ios::showpoint )
27         << setprecision( 10 ) << PI << endl;
28    return 0;
29 } // fin de la función main
```

*Salida del compilador Visual C++ de Microsoft*

```
  Valor local float de PI = 3.1415927410125732
Valor global double de PI = 3.14159265358979
  Valor float local de PI = 3.1415927410
```

**Figura 15.9** Uso del operador unario de resolución de alcance. (Parte 2 de 2.)



### Error común de programación 15.8

*Intentar acceder a una variable no global en un bloque externo por medio del operador unario de resolución de alcance es un error de sintaxis, si no existe una variable global con el mismo nombre que la variable en el bloque externo, y es un error de lógica si existe alguna.*



### Buena práctica de programación 15.4

*Evite utilizar variables con el mismo nombre para diferentes propósitos en un programa. Aunque esto está permitido en varios casos, puede resultar confuso.*

La instrucción (línea 19)

```
const float PI = static_cast< float >( ::PI );
```

incluye el operador **`static_cast< float>()`**, el cual crea una copia temporal en punto flotante de su operando (**`::PI`**). En C++, el operador **`static_cast`** reemplaza el estilo C de conversión de tipo que explicamos en capítulos anteriores. [*Nota:* C++ en realidad proporciona cuatro operadores de conversión de tipo, incluyendo **`static_cast`**, que en conjunto reemplazan al operador de tipo de estilo C. En este texto no explicamos los otros operadores de conversión de tipo. Las operaciones de conversión de tipo pueden volverse muy complejas y son propensas a errores, por lo que la comunidad de C++ sintió que al reemplazar el operador de conversión de estilo C, con nuevos operadores de conversión simplificaría las operaciones de conversión de tipo y reduciría los errores.]

En el capítulo 21 explicaremos con detalle las capacidades de formato de la figura 15.9, y aquí lo haremos brevemente. La llamada a **`setprecision( 20 )`** en la instrucción de salida (líneas 21 a 23)

```
cout << setprecision( 20 )
     << " Valor local float de PI = " << PI
     << "\nValor global double de PI = "<< ::PI << endl;
```

indica que la constante **`PI`** de tipo **`float`** va a imprimirse con veinte dígitos de *precisión* a la derecha del punto decimal (por ejemplo, **`3.1415927410125732`**). A esta llamada se le conoce como *manipulador parame-*

*trizado de flujo*. Si no se especifica la precisión, los valores de punto flotante normalmente se despliegan con seis dígitos de precisión (es decir, la *precisión predeterminada*), aunque en un momento veremos una excepción a esto.

La línea 25

```
cout << setw( 28 ) << "Valor local float de PI = "
```

llama a **setw( 28 )** para especificar que el siguiente valor de salida (es decir, **"Valor float local de PI = "**) se imprime en un *ancho de campo* de 28, es decir, el valor se imprime con al menos 28 posiciones de carácter. Si el valor a desplegar es menor que 28 posiciones de ancho, el valor se *justifica a la derecha* en el campo, de manera predeterminada. Si el valor a desplegar es menor que 28 posiciones de ancho, el ancho del campo se amplía para acomodar el valor completo.

Las líneas 26 y 27

```
<< setiosflags( ios::fixed | ios::showpoint )
<< setprecision( 10 ) << PI << endl;
```

indican que la constante **PI** se imprime como un *valor de punto fijo*, con un punto decimal (especificado con el manipulador de flujo **setiosflags( ios::fixed | ios::showpoint ))** y diez dígitos de precisión a la derecha del punto decimal (especificado con el manipulador **setprecision( 10 )**). Cuando se utiliza **setprecision** en un programa, el valor impreso se *redondea* para indicar el número de posiciones decimales, aunque el valor en memoria permanece intacto. Por ejemplo, los valores 87.945 y 67.543 se despliegan como 87.95 y 67.54, respectivamente, cuando a **setprecision** se le pasa un valor de **2**.

El manipulador de flujo **setiosflags( ios::fixed | ios::showpoint )** de la instrucción anterior establece dos opciones de formato de salida, a saber, ***ios::fixed*** y ***ios::showpoint***. El operador a nivel de bits **OR** incluyente (**|**), que explicamos en el capítulo 10, separa diversas opciones en una llamada a **setiosflags**. [*Nota:* Aunque las comas (**,**) con frecuencia se utilizan para separar una lista de elementos, éstas no pueden utilizarse con el manipulador de flujo **setiosflags**; de lo contrario, se establecerá sólo la última opción de la lista.] La opción **ios::fixed** ocasiona que se despliegue un valor de punto flotante en *formato de punto fijo* (lo contrario a la *notación científica*, la cual explicaremos en el capítulo 21). La opción **ios::showpoint** fuerza al punto decimal y a los ceros restantes a que se impriman, incluso si el valor es un número cerrado como 88.00. Sin la opción ***ios*::showpoint**, un valor como el anterior se imprimiría como 88, sin el punto decimal y sin los ceros de la derecha.

Los programas que utilizan estas llamadas deben contener la directiva de preprocesador (línea 9)

```
#include <iomanip>
```

Las líneas 11 a 13 especifican los nombres del archivo de encabezado **<iomanip>** que se utilizan en este programa. Observe que **endl** es un *manipulador no parametrizado de flujo*, y no requiere el archivo de encabezado **<iomanip>**. En el capítulo 21 explicaremos con mayor detalle las poderosas capacidades de formato de entrada/salida de **iomanip**.

## 15.10 Sobrecarga de funciones

C++ permite que se definan diversas funciones con el mismo nombre, mientras éstas tengan diferentes conjuntos de parámetros (al menos en lo que concierne a sus tipos). A esta capacidad se le llama *sobrecarga de funciones*. Cuando se llama a una función sobrecargada, el compilador de C++ selecciona la función adecuada examinando el número, los tipos y el orden de los argumentos en la llamada. La sobrecarga de funciones por lo general se utiliza para crear diversas funciones con el mismo nombre y que realizan tareas similares en diferentes tipos de datos.

**Buena práctica de programación 15.5**

*Sobrecargar funciones que realizan tareas muy relacionadas puede hacer que los programas sean más legibles y comprensibles.*

La figura 15.10 utiliza la función sobrecargada **cuadrado** para calcular el cuadrado de un **int** y el cuadrado de un **double**. En el capítulo 18, explicaremos cómo sobrecargar operadores para definir cómo deben

```
// Figura 15.10: fig15_10.cpp
// Uso de funciones sobrecargadas
#include <iostream>

using std::cout;
using std::endl;

int cuadrado( int x ) { return x * x; }

double cuadrado( double y ) { return y * y; }

int main()
{
   cout << “El cuadrado del entero  7 es “ << cuadrado( 7 )
        << “\nEl cuadrado del double 7.5 es “ << cuadrado( 7.5 )
        << endl;

   return 0;
} // fin de la función main
```

```
El cuadrado del entero  7 es 49
El cuadrado del double 7.5 es 56.25
```

**Figura 15.10** Uso de funciones sobrecargadas.

operar sobre objetos de tipos de datos definidos por el usuario. De hecho, hasta este punto hemos utilizado muchos objetos sobrecargados, incluyendo al operador de inserción de flujo **<<** y al operador de extracción de flujo **>>**. La sección 15.11 presenta las plantillas de funciones para generar funciones sobrecargadas que realizan tareas idénticas sobre tipos de datos diferentes.

Las funciones sobrecargadas se distinguen por sus *firmas*; una firma es una combinación del nombre de la función y sus tipos de parámetros. El compilador codifica cada identificador de función con el número y los tipos de sus parámetros (algunas veces conocido como *separación de nombre* o *decoración de nombre*) para permitir la *vinculación de tipo segura*. La vinculación de tipo segura garantiza que se llama a la función sobrecargada adecuada y que los argumentos coinciden con los parámetros. El compilador detecta y reporta los errores de vinculación.

**Error común de programación 15.9**

*Crear funciones sobrecargadas con listas de parámetros idénticas y diferentes tipos de retorno, es un error de sintaxis.*

El compilador sólo utiliza las listas de parámetros para distinguir entre funciones del mismo nombre. Las funciones sobrecargadas necesitan no tener el mismo número de parámetros. Los programadores deben ser cautelosos cuando sobrecarguen funciones con parámetros predeterminados, ya que esto podría ocasionar ambigüedades.

## 15.11 Plantillas de funciones

Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos. Si la lógica de un programa y las operaciones son idénticas para cada tipo de dato, esto se podría realizar de manera más compacta y conveniente por medio de *plantillas de funciones*. El programador escribe una sola definición de plantilla de función. Dado que los tipos de argumentos se proporcionan en las llamadas a función, C++ genera *plantillas de funciones* separadas para manejar adecuadamente cada tipo de llamada. Entonces, al definir una sola plantilla de función se define a toda una familia de soluciones. En el capítulo 22, presentaremos una característica relacionada de C++ llamada plantillas de funciones.

Todas las definiciones de plantillas de función comienzan con la palabra reservada **template**, seguida por una lista de tipos de parámetros formales para la plantilla de función encerrada entre corchetes angulares (**<** y **>**). Todo tipo de parámetro formal es precedido por la palabra reservada **typename** o por la palabra reservada **class**. Los *tipos de parámetros formales* son tipos integrados o definidos por el usuario que se utilizan para especificar los tipos de los argumentos de la función, para especificar el tipo de retorno de la función, y para declarar variables dentro del cuerpo de la definición de la función.

La siguiente definición de plantilla de función también se utiliza en la figura 15.11 (líneas 9 a 21):

```
template < class T > //o template < typename T>
T maximo( T valor1, T valor2, T valor3 )
{
   T max = valor1;

   if ( valor2 > max )
      max = valor2;

   if (valor3 > max )
      max = valor3;

   return max;
} // fin de la plantilla de función maximo
```

Esta plantilla de función declara un solo tipo de parámetro formal **T** como el tipo de dato a probar por la función **maximo**. Cuando el compilador detecta una llamada a **maximo** en el código fuente del programa, el tipo del dato pasado a **maximo** se sustituye con **T** a lo largo de la definición de la plantilla, y C++ crea una función completa para determinar el máximo de los tres valores del tipo de dato especificado. Después, se compila la función recientemente creada. Por lo tanto, las plantillas realmente son un medio de generación de código. En la figura 15.11, se crean tres funciones; una espera tres valores **int**, otra espera tres valores **double**, y la otra espera tres valores **char**. La plantilla de función creada para el tipo **int** es:

```
int maximo( int valor1, int valor2, int valor3 )
{
   int max = valor1;

   if ( valor2 > max )
      max = valor2;

   if ( valor3 > max )
      max = valor3;

   return max;
}
```

El nombre de un tipo de parámetro debe ser único en la lista de parámetros formales de una definición de plantilla en particular. La figura 15.11 ilustra el uso de una plantilla de función llamada **maximo** para determinar el mayor de tres valores **int**, tres valores **double**, y tres valores **char**.

```
1  // Figura 15.11: fig15_11.cpp
2  // Uso de una plantilla de función
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  template < class T >
10 T maximo( T valor1, T valor2, T valor3 )
```

**Figura 15.11** Uso de una plantilla de función. (Parte 1 de 2.)

```
{
   T max = valor1;

   if ( valor2 > max )
      max = valor2;

   if ( valor3 > max )
      max = valor3;

   return max;
} // fin de la plantilla de función maximo

int main()
{
   int int1, int2, int3;

   cout << "Introduzca tres valores enteros: ";
   cin >> int1 >> int2 >> int3;
   cout << "El valor entero maximo es : "
        << maximo( int1, int2, int3 );          // versión int

   double doble1, doble2, doble3;

   cout << "\nIntroduzca tres valores double: ";
   cin >> doble1 >> doble2 >> doble3;
   cout << "El valor double maximo es: "
        << maximo( doble1, doble2, doble3 );   // versión double

   char char1, char2, char3;

   cout << "\nIntroduzca tres caracteres: ";
   cin >> char1 >> char2 >> char3;
   cout << "El valor caracter maximo es: "
        << maximo( char1, char2, char3 )       // versión char
        << endl;

   return 0;
} // fin de la función main
```

```
Introduzca tres valores enteros: 1 2 3
El valor entero maximo es : 3
Introduzca tres valores double: 3.3 2.2 1.1
El valor double maximo es: 3.3
Introduzca tres caracteres: A C B
El valor caracter maximo es: C
```

**Figura 15.11** Uso de una plantilla de función. (Parte 2 de 2.)



### Error común de programación 15.10

*No colocar la palabra reservada* **`class`** *o la palabra reservada* **`typename`** *antes de cada tipo de parámetro correspondiente a una plantilla de función, es un error de sintaxis.*

## RESUMEN

- La biblioteca estándar de C++ contiene muchas funciones y clases que los programadores pueden aprovechar en sus programas.
- Los comentarios de una sola línea de C++ comienzan con `//`.

- La línea **#include<iostream>** le indica al preprocesador de C++ que incluya en el programa el contenido del archivo de encabezado de flujo de entrada/salida. Este archivo contiene la información necesaria para compilar los programas que utilizan **std::cin**, **std::cout**, **std::endl**, y los operadores **<<** y **>>**.
- El objeto de flujo de salida **std::cout**, normalmente conectado a la pantalla, se utiliza para desplegar datos. Es posible desplegar múltiples datos, concatenando operadores de inserción de flujo (**<<**).
- El objeto de flujo de entrada **std::cin**, normalmente conectado al teclado, se utiliza para introducir datos. Es posible introducir múltiples datos, concatenando operadores de extracción de flujo (**>>**).
- Las instrucciones

```
using std::cout;
using std::cin;
using std::endl;
```

  utilizan instrucciones **using** que nos ayudan a eliminar la necesidad de repetir el prefijo **std::**. Una vez que incluimos estas instrucciones **using**, podemos escribir **cout** en lugar de **std::cout**, **cin** en lugar de **std::cin**, y **endl** en lugar de **std::endl**, en el resto del programa.
- Las funciones **inline** eliminan la sobrecarga de llamadas a función. El programa utiliza la palabra reservada **inline** para aconsejar al compilador que genere código en línea (cuando sea posible) para minimizar las llamadas a funciones. El compilador puede elegir ignorar la solicitud **inline**.
- C++ proporciona la palabra reservada **bool** para representar valores **booleanos**. A un **bool** se le puede asignar un **0**, un valor diferente de cero, la palabra reservada **true** o la palabra reservada **false**.
- C++ ofrece una forma directa de realizar llamadas por referencia, por medio de parámetros de referencia. Para indicar que un parámetro de función pasa por referencia, coloque un amperson después del tipo del parámetro en el prototipo de la función. En la llamada a la función, mencione a la variable por su nombre y ésta pasará en una llamada por referencia. En la función llamada, mencionar a la variable por su nombre local, en realidad se hace referencia a la variable original de la función que realiza la llamada. Por lo tanto, la variable original puede ser modificada directamente por la función llamada.
- Los parámetros de referencias también pueden crearse para uso local, como un alias de otras variables dentro de una función. Las variables de referencia deben inicializarse en sus declaraciones, y no pueden reasignarse como alias de otras variables. Una vez que se declara una variable de referencia como un alias de otra variable, todas las operaciones que supuestamente se realizan sobre el alias, en realidad se realizan sobre la variable.
- C++ permite que el programador especifique argumentos predeterminados de funciones y sus valores predeterminados. Si un argumento predeterminado se omite en una llamada a una función, se utiliza el valor predeterminado de ese argumento. Los argumentos predeterminados deben ser los argumentos que se encuentren más a la derecha de la lista de parámetros de la función. Los argumentos predeterminados deben especificarse con la primera ocurrencia del nombre de la función. Los valores predeterminados pueden ser constantes, variables globales o llamadas a función.
- C++ permite al programador especificar una lista de parámetros vacía, por medio de la palabra reservada **void**, o simplemente dejando la lista de parámetros vacía.
- El operador unario de resolución de alcance (**::**) permite a un programa acceder a una variable global, cuando una variable local del mismo tipo se encuentra al alcance.
- El operador unario de conversión de tipo, **static_cast< float >( )**, crea una copia temporal de punto flotante de su operando.
- El manipulador de flujo **setw** especifica que un valor debe imprimirse en un campo de tamaño especificado, y se justifica a la derecha de manera predeterminada. Si el valor a desplegar es mayor que el ancho de campo especificado, el ancho de campo se amplía para acomodar el valor completo. Los programas que utilizan esta llamada, deben contener la directiva de preprocesador **#include< iomanip >**.
- El manipulador de flujo **setiosflags( ios::fixed | ios::showpoint )** establece dos opciones de formato de salida, a saber, **ios::fixed** y **ios::showpoint**. El operador a nivel de bits **OR** incluyente (**|**) separa diversas opciones en una llamada a **setiosflags**. La opción **ios::fixed** ocasiona que se despliegue un valor de punto flotante en un *formato de punto fijo* (lo contrario a la *notación científica*). La opción **ios::showpoint** fuerza al punto decimal y a los ceros restantes a que se impriman, incluso si el valor es un número cerrado. Los programas que utilizan estas llamadas, deben contener la directiva de preprocesador **#include< iomanip >**.
- Es posible definir diversas funciones con el mismo nombre pero con diferentes tipos de parámetros. A esto se le llama sobrecarga de funciones. Cuando se llama a una función sobrecargada, el compilador selecciona la función adecuada, examinando el número y los tipos de los argumentos en la llamada.

- Las funciones sobrecargadas pueden tener diferentes valores de retorno, y deben tener diferentes listas de parámetros. Dos funciones que difieren sólo en el tipo de retorno, producirán un error de compilación.
- Las plantillas de funciones permiten la creación de funciones que realicen las mismas operaciones sobre diferentes tipos de datos, pero la plantilla de función se define sólo una vez.

## TERMINOLOGÍA

**//** para comentarios en C++
**<iomanip>**
**<iostream>**
alcance
alcance de función
alias
ancho de campo
archivo de encabezado
archivos de encabezado de la biblioteca estándar
argumento de una llamada a función
argumentos predeterminados de una función
biblioteca estándar de C++
**bool**
C++
**cin**
clase
**class**
componente
copia de un valor
**cout**
decoración de nombre
**endl**
espacio de nombre
**false**
firma
firma de función
flujo de entrada estándar
flujo de salida estándar
flujos
formato de punto fijo
función **inline**
función llamada
función que llama
función que realiza una llamada
función **template**
**ios::fixed**
**ios::showpoint**
llamada a función
llamada por referencia
llamada por valor
**lvalue**
manipulador de flujo
manipulador no parametrizado de flujo
manipulador parametrizado de flujo
notación científica
operación unario de resolución de alcance (**::**)
operador de extracción de flujo (**>>**)
operador de inserción de flujo (**<<**)
parámetro de referencia
parámetro en una definición de función
prefijo **std::**
programación orientada a objetos
referencia indefinida
**rvalue**
separación de nombre
**setiosflags**
**setprecision**
**setw**
sobrecarga
sobrecarga de funciones
**static_cast**
sufijo amperson (**&**)
**template**
tipo de referencia
**true**
**typename**
**using**
valor de punto fijo
variable global
variable local
vinculación de tipo segura

## ERRORES COMUNES DE PROGRAMACIÓN

**15.1** Omitir el tipo de valor de retorno en una definición de función de C++, es un error de sintaxis.

**15.2** Los parámetros de referencia se mencionan sólo por nombre en el cuerpo de la función llamada, por lo que el programador podría tratar inadvertidamente a los parámetros de referencia como parámetros de una llamada por valor. Esto puede ocasionar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función que hace la llamada.

**15.3** No inicializar una variable de referencia cuando ésta se declara, es un error de sintaxis.

**15.4** Intentar reasignar una referencia previamente declarada para que sea un alias de otra variable, es un error lógico. El valor de la otra variable simplemente se asigna a la ubicación para la cual, la referencia ya es un alias.

**15.5** Declarar varias referencias en una instrucción, mientras se asume que el **&** se distribuye a lo largo de una lista de nombres de variables separados por comas. Para declarar las variables **x**, **y** y **z** como referencias a enteros, utilice la notación **int &x = a, &y = b, &z = c;** en lugar de utilizar la notación incorrecta **int& x = a, y = b, z = c;** o la otra notación común incorrecta **int &x, y, z;**.

**15.6** Devolver un apuntador o referencia a una variable automática en una función llamada, es un error lógico. Algunos compiladores despliegan una advertencia, cuando esto ocurre en un programa.

**15.7** Especificar e intentar utilizar un argumento predeterminado que no sea el que se encuentra más a la derecha (mientras a los demás argumentos no se les asigne simultáneamente valores predeterminados) es un error de sintaxis.

**15.8** Intentar acceder a una variable no global en un bloque externo por medio del operador unario de resolución de alcance es un error de sintaxis, si no existe una variable global con el mismo nombre que la variable en el bloque externo, y es un error de lógica si existe alguna.

**15.9** Crear funciones sobrecargadas con listas de parámetros idénticas y diferentes tipos de retorno, es un error de sintaxis.

**15.10** No colocar la palabra reservada `class` o la palabra reservada `typename` antes de cada tipo de parámetro correspondiente a una plantilla de función, es un error de sintaxis.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**15.1** Si prefiere colocar declaraciones al principio de una función, separe dichas declaraciones de las instrucciones ejecutables de esa función con una línea en blanco, para resaltar el lugar en donde terminan las declaraciones y en donde comienzan las instrucciones ejecutables.

**15.2** El calificador `inline` debe utilizarse sólo con pequeñas funciones que se utilicen con frecuencia.

**15.3** Utilizar argumentos predeterminados simplifica la escritura de llamadas a función. Sin embargo, algunos programadores sienten que especificar explícitamente todos los argumentos es más claro.

**15.4** Evite utilizar variables con el mismo nombre para diferentes propósitos en un programa. Aunque esto está permitido en varios casos, puede resultar confuso.

**15.5** Sobrecargar funciones que realizan tareas muy relacionadas puede hacer que los programas sean más legibles y comprensibles.

## *TIPS DE RENDIMIENTO*

**15.1** Utilizar las funciones y clases de la biblioteca estándar en lugar de escribir las suyas, puede mejorar el rendimiento del programa, ya que este software está cuidadosamente escrito para que funcione correcta y eficientemente.

**15.2** Utilizar funciones `inline` puede reducir el tiempo de ejecución, pero incrementar el tamaño del programa.

**15.3** Una desventaja de las llamadas por valor es que, si se está pasando un elemento de datos grande, copiar dicho elemento puede implicar demasiado tiempo de ejecución.

**15.4** Una llamada por referencia es buena por motivos de rendimiento, ya que ésta elimina la sobrecarga de copiar grandes cantidades de datos.

**15.5** Para pasar objetos grandes, utilice un parámetro de referencia constante para simular la apariencia y seguridad de una llamada por valor y para evitar la sobrecarga de pasar una copia de ese gran objeto.

## *TIPS DE PORTABILIDAD*

**15.1** Utilizar las funciones y clases de la biblioteca estándar en lugar de escribir las suyas, puede mejorar la portabilidad del programa, ya que este software está incluido en casi todas las implementaciones de C++.

**15.2** En C++, el significado de una lista de parámetros vacía es dramáticamente diferente que en C. En C, significa que toda verificación de argumentos está deshabilitada (es decir, la llamada de función puede pasar cualquier argumento que desee). En C++, significa que la función no toma argumentos. Por lo tanto, los programas en C que utilizan estas características podrían reportar errores de sintaxis cuando se compilan en C++.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**15.1** Utilice un "método de construcción en bloques" para crear programas. Evite reinventar la rueda. Utilice piezas existentes en donde sea posible; a esto se le llama "reutilización de software", y es básica para la programación orientada a objetos.

**15.2** Cuando programe en C++, normalmente utilice los siguientes bloques de construcción: clases y funciones de la biblioteca estándar de C++, clases y funciones que genere usted mismo, y clases y funciones de otras bibliotecas populares provistas por fabricantes de terceros.

**15.3** En C++ son necesarios los prototipos de función. Utilice las directivas de preprocesador `#include` para obtener los prototipos de función de la biblioteca estándar. También utilice `#include` para obtener los archivos de encabezado que contienen los prototipos de función utilizados por usted y/o por los miembros de su grupo.

**15.4** Cualquier modificación a una función `inline` podría requerir que todos los clientes de dicha función se recompilaran. Esto puede ser importante en algunas situaciones de desarrollo de programas y de mantenimiento.

**15.5** Una llamada por referencia puede debilitar la seguridad, ya que la función llamada puede corromper los datos de la función que la llamó.

**15.6** Por razones combinadas de claridad y rendimiento, muchos programadores en C++ prefieren que los argumentos modificables sean pasados por medio de apuntadores, que los argumentos pequeños no modificables pasen por medio de una llamada por valor, y que los argumentos grandes no modificables pasen por medio de referencias a constantes.

## EJERCICIOS DE AUTOEVALUACIÓN

**15.1** Responda cada una de las siguientes preguntas:

a) En C++, es posible tener diversas funciones con el mismo nombre, que operen sobre diferentes tipos y/o números de argumentos. A esto se le llama __________ de funciones.

b) El __________ permite acceder a una variable global con el mismo nombre que una variable en el alcance actual.

c) Una __________ de función permite que se defina a una sola función para que realice una tarea sobre muchos tipos de datos diferentes.

**15.2** ¿Por qué un prototipo de función contendría una declaración de tipo de parámetro como `double&`?

**15.3** (Verdadero/falso.) Todas las llamadas en C++ se realizan por valor.

**15.4** Escriba un programa completo que utilice una función `inline` llamada `volumenEsfera` que pida al usuario el radio de una esfera y que calcule e imprima el volumen de dicha esfera, por medio de la asignación `volumen = ( 4.0 / 3 ) * 3.14159 * pow( radio, 3 )`.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**15.1** a) Sobrecarga. b) Operador unario de resolución de alcance (`::`). c) Plantilla.

**15.2** El programador declara un parámetro de referencia de tipo `double` para obtener acceso a la variable argumento original a través de una llamada por referencia.

**15.3** Falso. C++ permite directamente las llamadas por referencia a través de los parámetros de referencia, además de hacerlo por medio de apuntadores.

**15.4** Vea el siguiente código.

```
// ej15_04.cpp
// Función inline que calcula el volumen de una esfera
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <cmath>

const double PI = 3.14159;

inline double volumenEsfera( const double r )
   { return 4.0 / 3.0 * PI * pow( r, 3 ); }

int main()
{
   double radio;

   cout << “Introduzca la longitud del radio de su esfera: “;
   cin >> radio;
   cout << “El volumen de la esfera con radio “ << radio <<
           “ es “ << volumenEsfera( radio) << endl;
   return 0;
} // fin de la función main
```

## EJERCICIOS

**15.5** Escriba un programa en C++ que utilice una función `inline` llamada `areaCirculo` que pida al usuario el radio de un círculo, y que calcule e imprima el área de dicho círculo.

**15.6** Escriba un programa completo en C++ con las dos funciones alternativas especificadas abajo, en donde cada una de ellas simplemente triplica la variable cuenta definida en `main`. Después compare y contraste los dos métodos. Estas dos funciones son:

a) La función `tripleLlamadaPorValor` que pasa una copia de cuenta por medio de una llamada por valor, que triplica la copia y devuelve el nuevo valor.

b) La función `triplePorReferencia` que pasa cuenta con una verdadera llamada por referencia, a través de un parámetro de referencia, y que triplica la copia original de cuenta por medio de su alias (es decir, el parámetro de referencia).

**15.7** ¿Cuál es el propósito del operador unario de resolución de alcance?

**15.8** Escriba un programa que utilice una plantilla de función llamada `min`, para determinar el menor de dos argumentos. Evalúe el programa utilizando un par de enteros, uno de caracteres y uno de punto flotante.

**15.9** Escriba un programa que utilice una plantilla de función llamada `max`, para determinar el mayor de tres argumentos. Evalúe el programa utilizando un par de enteros, uno de caracteres y uno de punto flotante.

**15.10** Determine si los siguientes segmentos de programa contienen errores. Por cada error, explique cómo puede corregirlo. [*Nota:* Para un segmento de un programa en particular, es posible que no existan errores.]

a)
```
template < class A >
int suma(int num1, int num2, int num3 )
   { return num1 + num2 + num3 }
```

b)
```
void imprimeResultados( int x, int y )
{
   cout << "La suma es " << x + y << '\n';
   return x + y;
}
```

c)
```
template < A >
A producto( A num1, A num2, A num3 )
{
   return num1 * num2 * num3;
}
```

d)
```
double cubo( int );
int cubo( int );
```

# 16

# Clases y abstracción de datos en C++




## Objetivos

- Comprender los conceptos de ingeniería de software con respecto al encapsulamiento y al ocultamiento de datos.
- Comprender las nociones de la abstracción de datos y de los tipos de datos abstractos (ADTs).
- Crear ADTs en C++, es decir, clases.
- Comprender cómo crear, cómo utilizar y cómo destruir objetos de clases.
- Controlar el acceso a los datos y a las funciones miembro de los objetos.
- Comenzar a apreciar el valor de la orientación a objetos.

*Mis objetivos, todos sublimes,*
*deberé conseguirlos a tiempo.*
W. S. Gilbert

*¿Es éste un mundo en el cual esconder virtudes?*
William Shakespeare

*Tus sirvientes públicos te sirven correctamente.*
Adlai Stevenson

*Los rostros privados en lugares públicos*
*son más sabios y amables*
*que los rostros públicos en lugares privados.*
W. H. Auden

## Plan general



## 16.1 Introducción

Ahora comenzaremos con nuestra introducción a la orientación a objetos en C++. En los capítulos 1 a 14 presentamos la programación estructurada en C. Los objetos que construiremos en C++ estarán compuestos parcialmente por piezas de programación estructurada.

Revisemos brevemente algunos conceptos clave y la terminología de la orientación a objetos. La programación orientada a objetos (POO) *encapsula* datos (atributos) y funciones (comportamiento) en paquetes llamados *clases*; los datos y las funciones de una clase se encuentran íntimamente ligados entre sí. Una clase es como un anteproyecto. A partir de un anteproyecto, un constructor puede construir una casa. A partir de una clase, un programador puede crear un objeto. Un anteproyecto puede reutilizarse muchas veces para hacer varias casas. Una clase puede reutilizarse muchas veces para crear muchos objetos de la misma clase. Las clases tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos de una clase pueden saber cómo comunicarse entre sí, a través de *interfaces* bien definidas, por lo general a las clases no se les permite saber cómo se implementan otras clases; los detalles de implementación están ocultos dentro de las mismas clases. Con toda seguridad es posible conducir un automóvil de manera efectiva sin conocer los detalles de cómo funcionan internamente los sistemas del motor, de la transmisión y del escape. Veremos por qué el ocultamiento de información es tan importante para la buena ingeniería de software.

En C y en otros *lenguajes de programación por procedimientos*, la programación tiende a ser *orientada a acciones*, mientras que en C++ el ideal es programar con *orientación a objetos*. En C, la unidad de programación es la *función*. En C++, la unidad de programación es la *clase*, a partir de la cual se generan las *instancias* de todos los objetos (es decir, se crean).

Los programadores en C se concentran en escribir funciones. Los conjuntos de acciones que realizan alguna tarea se agrupan en funciones, y las funciones se agrupan para formar programas. Ciertamente, los datos son importantes en C, pero la idea es que los datos existen primordialmente para apoyar las acciones que realizan las funciones. En la especificación de un sistema, los *verbos* ayudan al programador en C a determinar el conjunto de funciones que trabajarán juntas para implementar el sistema.

Los programadores en C++ se concentran en crear sus propios *tipos definidos por el programador* llamados *clases*. A las clases también se les denomina *tipos definidos por el programador*. Cada clase contiene datos, así como un conjunto de funciones que manipulan estos datos. A los datos que componen una clase se les

llama *datos miembro*. A las funciones que componen una clase se les llama *funciones miembro* (o *métodos*, en otros lenguajes orientados a objetos). Así como a una instancia de un tipo de dato predefinido, tal como **`int`**, se le llama *variable*, a una instancia de un tipo de dato definido por el usuario (es decir, a la instancia de una clase) se le llama *objeto*. [En la comunidad de C++, los términos variable y objeto a menudo se utilizan de manera indistinta.] El foco de atención en C++ se centra en las clases, en lugar de las funciones. Los *sustantivos* que se encuentran en las especificaciones de un sistema ayudan al programador en C++ a determinar el conjunto de clases que utilizará para crear los objetos que trabajarán juntos para implementar el sistema.

Las clases en C++ son la evolución natural de la idea de C con respecto a **`struct`** que explicamos en el capítulo 10. Recuerde que una **`struct`** es una colección de variables (datos) relacionadas, mientras que una clase contiene tanto variables (*datos miembro*) como las funciones que manipulan dichos datos (*funciones miembro*). En la siguiente sección desarrollaremos el tipo de dato abstracto **`Hora`** como una clase de C++. Veremos que las clases proporcionan una manera sólida de describir nuevos tipos de datos abstractos.

## 16.2 Implementación del tipo de dato abstracto Hora mediante una clase

Las clases permiten al programador modelar objetos que tienen *atributos* (representados como *datos miembro*) y *comportamientos* u *operaciones* (representadas como *funciones miembro*). En C++, los tipos que contienen datos miembro y funciones miembro se definen mediante la palabra reservada **`class`**.

Algunas veces, en otros lenguajes de programación orientada a objetos, a las funciones miembro se les denomina *métodos* y se invocan en respuesta a los *mensajes* que se envían al objeto. Un mensaje corresponde a una llamada a una función miembro enviada de un objeto a otro, o enviada desde una función hacia un objeto.

Una vez que se define una clase, el nombre de la clase se vuelve un nombre de tipo, el cual puede utilizarse para declarar objetos de dicha clase. La figura 16.1 contiene una definición sencilla para la clase **`Hora`**.

Nuestra definición de la clase **`Hora`** comienza con la palabra reservada **`class`**. El *cuerpo* de la definición de la clase está delimitado con las llaves izquierda y derecha (**`{`** y **`}`**). La definición de la clase termina con un punto y coma. Nuestra definición de la clase **`Hora`** contiene los miembros de tipo entero **`hora`**, **`minuto`** y **`segundo`**.



**Error común de programación 16.1**

*Olvidar el punto y coma al final de una definición de clase (o de una estructura), es un error de sintaxis.*

Las etiquetas ***`public:`*** y ***`private:`*** se llaman *especificadores de acceso a miembros*. Cualquier dato o función miembro declarada después del especificador **`public`** (y antes del siguiente especificador de acceso a miembro) es accesible desde cualquier parte el programa en la que el objeto **`Hora`** se encuentre al alcance. Cualquier dato o función miembro que se declara después del especificador de acceso a miembros **`private`** (y hasta el siguiente especificador de acceso a miembros) sólo es accesible a funciones miembro de la clase. Los especificadores de acceso a miembros terminan siempre con dos puntos (**`:`**), y pueden aparecer varias veces y en cualquier orden dentro de la definición de la clase. Durante el resto del libro, haremos referencia a los especificadores de acceso a miembros como **`public`** o **`private`** (sin dos puntos). En el capítulo 19 presentaremos el tercer especificador de acceso a miembros, llamado **`protected`**, al estudiar la herencia y el papel que ésta juega en la programación orientada a objetos.

```
class Hora {
public:
   Hora();
   void estableceHora( int, int, int );
   void imprimeMilitar();
   void imprimeEstandar();
private:
   int hora;     // 0 - 23
   int minuto;   // 0 - 59
   int segundo;   // 0 - 59
}; // fin de la clase Hora
```

**Figura 16.1** Una definición sencilla de la clase **`Hora`**.



**Buena práctica de programación 16.1**

*Para mayor claridad, utilice cada especificador de acceso a miembros una sola vez dentro de la definición de la clase. Primero coloque los elementos **`public`** en donde sean fáciles de localizar.*

La definición de la clase contiene prototipos para las siguientes cuatro funciones miembro después del especificador de acceso a miembros **`public`**: **`Hora`**, **`estableceHora`**, **`imprimeMilitar`** e **`imprimeEstandar`**. Éstas son las *funciones miembro* ***`public`*** de la clase (también conocidas como *servicios públicos*, *comportamientos públicos* o *interfaz* de la clase). Estas funciones serán utilizadas por los *clientes* (por ejemplo, porciones de un programa que son usuarios) de la clase para manipular los datos de dicha clase. Los datos miembro de la clase permite el envío de los *servicios* que la clase proporciona a sus clientes, a través de sus funciones miembro. Estos servicios permiten al código cliente interactuar con un objeto de dicha clase.

Observe la función miembro que tiene el mismo nombre que la clase; a ésta se le denomina función *constructor* de la clase. Un constructor es una función miembro especial que inicializa los datos miembro de un objeto de la clase. A un constructor de la clase se le invoca cuando el programa crea un objeto de dicha clase. Veremos que es común tener varios constructores para una clase; esto se lleva a cabo a través de la sobrecarga de funciones. Observe que no es posible especificar ningún tipo de retorno para el constructor.

**Error común de programación 16.2**

*Especificar un tipo o un valor de retorno para un constructor, es un error de sintaxis.*

Los tres miembros enteros aparecen después del especificador de acceso a miembros **`private`**. Esto indica que los datos miembro de la clase son accesibles sólo para las funciones miembro (y, como veremos en el siguiente capítulo, son "amigos") de la clase. Por lo tanto, solamente se puede acceder a los datos miembro de la clase **`Hora`** mediante las cuatro funciones, cuyos prototipos aparecen en la definición de la clase. Por lo general, los datos miembro se listan en la parte privada de la clase, y las funciones miembro se listan en la porción pública. Como veremos más adelante, es posible tener funciones miembro **`private`** y datos **`public`**; el uso de datos **`public`** no es común y se considera como una mala práctica de programación.

Una vez que se define la clase, es posible utilizarla como un tipo dentro de las declaraciones de objetos, arreglos y apuntadores, de la siguiente manera:

```
Hora atardecer,              // objeto de tipo Hora
    arregloDeHoras[ 5 ],      // arreglo de objetos de tipo Hora
    *apuntadorAHora,          // apuntador a un objeto de tipo Hora
    &horaCenar = atardecer;   // referencia a un objeto de tipo Hora
```

El nombre de la clase se convierte en un especificador de tipo. Puede haber tantos objetos de una clase, como variables de tipo **`int`**. El programador puede crear nuevos tipos de clases, como sea necesario. Ésta es una de las razones por las que C++ es un *lenguaje extensible*.

La figura 16.2 utiliza la clase **`Hora`**. El programa crea una instancia de la clase **`Hora`** llamada **`h`**. Cuando se crea la instancia del objeto, se llama al constructor **`Hora`** para inicializar en 0 a cada dato miembro **`private`**. Después, se imprime la hora en formato militar y en formato estándar para confirmar que los miembros se inicializaron de manera apropiada. Entonces, con el uso de la función miembro **`estableceHora`** se establece la hora y se imprime de nuevo en ambos formatos. Luego, **`estebleceHora`** intenta establecer los datos miembro con valores inválidos, y se imprime de nuevo la hora en ambos formatos.

```
// Figura 16.2: fig16_02.cpp
// Clase Hora.
#include <iostream>

using std::cout;
using std::endl;

// Definición del tipo de dato abstracto (ADT) Hora
class Hora {
```

**Figura 16.2** Implementación del tipo de dato abstracto **`Hora`** como una clase. (Parte 1 de 3.)

```
public:
   Hora();                             // constructor
   void estableceHora( int, int, int ); // establece hora, minuto, segundo
   void imprimeMilitar();              // imprime la hora en formato
                                          militar
   void imprimeEstandar();             // imprime la hora en formato
                                          estándar
private:
   int hora;     // 0 - 23
   int minuto;   // 0 - 59
   int segundo;  // 0 - 59
}; // fin de la clase Hora

// El constructor Hora inicializa en cero a cada dato miembro.
// Garantiza que todos los objetos de Hora inician en un estado consistente.
Hora::Hora() { hora = minuto = segundo = 0; }

// Establece un nuevo valor de Hora por medio de la hora militar. Realiza
   verificaciones
// de validación de los valores de los datos. Establece en cero a los
   valores inválidos.
void Hora::estableceHora( int h, int m, int s )
{
   hora = ( h >= 0 && h < 24 ) ? h : 0;
   minuto = ( m >= 0 && m < 60 ) ? m : 0;
   segundo = ( s >= 0 && s < 60 ) ? s : 0;
} // fin de la función estableceHora

// Imprime Hora en formato militar
void Hora::imprimeMilitar()
{
   cout << ( hora < 10 ? “0” : “” ) << hora << “:”
        << ( minuto < 10 ? “0” : “” ) << minuto;
} // fin de la función imprimeMilitar

// Imprime Hora en formato estándar
void Hora::imprimeEstandar()
{
   cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
        << “:” << ( minuto  < 10 ? “0” : “” ) << minuto
        << “:” << ( segundo < 10 ? “0” : “” ) << segundo
        << ( hora < 12 ? “ AM” : “ PM” );
} // fin de la función imprimeEstandar

// Controlador para probar la clase simple Hora
int main()
{
   Hora h;  // instancia el objeto h de la clase Hora

   cout << “La hora militar inicial es “;
   h.imprimeMilitar();
   cout << “\nLa hora estandar inicial es “;
   h.imprimeEstandar();

   h.estableceHora( 13, 27, 6 );
   cout << “\n\nLa hora militar despues de estableceHora es “;
```

**Figura 16.2** Implementación del tipo de dato abstracto **Hora** como una clase. (Parte 2 de 3.)

```
    h.imprimeMilitar();
    cout << "\nLa hora estandar despues de estableceHora es ";
    h.imprimeEstandar();

    h.estableceHora( 99, 99, 99 );  // intenta establecer valores inválidos
    cout << "\n\nDespues de intentar establecer valores invalidos:"
         << "\nHora militar: ";
    h.imprimeMilitar();
    cout << "\nHora estandar: ";
    h.imprimeEstandar();
    cout << endl;
    return 0;
} // fin de la función main
```

```
La hora militar inicial es 00:00
La hora estandar inicial es 12:00:00 AM

La hora militar despues de estableceHora es 13:27
La hora estandar despues de estableceHora es 1:27:06 PM

Despues de intentar establecer valores invalidos:
Hora militar: 00:00
Hora estandar: 12:00:00 AM
```

**Figura 16.2** Implementación del tipo de dato abstracto **Hora** como una clase. (Parte 3 de 3.)

De nuevo, observe que los datos miembro **hora**, **minuto** y **segundo** están precedidos por el especificador de acceso a miembros **private**. Por lo general, los datos miembro **private** de una clase no son accesibles fuera de la clase. (Otra vez, en el capítulo 17 veremos que los amigos de una clase pueden acceder a sus miembros privados.) Aquí, la filosofía es que la representación de los datos que se utiliza en la clase no les concierne a los clientes de la clase. Por ejemplo, sería perfectamente razonable para la clase representar la hora de manera interna como el número de segundos a partir de la medianoche. Los clientes podrían utilizar las mismas funciones miembro públicas y obtener los mismos resultados sin darse cuenta de esto. En este sentido, se dice que la implementación de una clase se *oculta* a sus clientes. Dicho *ocultamiento de información* promueve la modificación del programa y simplifica la percepción del cliente con respecto a una clase.

### Observación de ingeniería de software 16.1

*Los clientes de una clase la utilizan sin conocer los detalles internos acerca de la manera en que se implementa. Si se modifica la implementación de una clase (por ejemplo, para mejorar el rendimiento), debido a que la interfaz de la clase permanece constante, el código fuente cliente de la clase no requiere modificación alguna (aunque el código cliente deberá compilarse de nuevo). Esto hace mucho más fácil la modificación de sistemas.*

En este programa, el constructor **Hora** inicializa en 0 a los datos miembro (es decir, el equivalente militar de las 12 AM). Esto garantiza que cuando se crea el objeto, éste se encuentra en un estado consistente. No es posible almacenar valores no válidos en los datos miembro de un objeto **Hora**, debido a que se invoca al constructor cuando se crea dicho objeto y a que mediante la función **estableceHora** se escrutan todos los intentos subsecuentes de modificación de los datos miembro por parte del cliente.



### Observación de ingeniería de software 16.2

*Por lo general, las funciones miembro son más pequeñas que las que se encuentran en programas no orientados a objetos, debido a que los datos almacenados en los datos miembro se validan por medio del constructor, o por medio de las funciones miembro que almacenan los nuevos datos. Debido a que los datos ya se encuentran en el objeto, las llamadas a las funciones miembro a menudo se hacen sin argumentos, o al menos tienen menos argumentos que las típicas llamadas a funciones en lenguajes no orientados a objetos. Por lo tanto, las llamadas, las definiciones de función y los prototipos de las funciones son más cortos.*

Observe que los datos miembro de una clase no pueden inicializarse en la parte en la que están declarados dentro del cuerpo de la clase. Estos datos miembro deben inicializarse mediante el constructor de la clase, o pueden ser valores asignados mediante funciones "establecer".

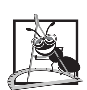


### Error común de programación 16.3

*Intentar inicializar explícitamente un dato miembro de una clase dentro de la definición de la clase, es un error de sintaxis.*

A una función con el mismo nombre de la clase, pero precedida por el *carácter tilde* (**~**) se le denomina *destructor* de la clase (este ejemplo no incluye explícitamente un destructor, de modo que la implementación de C++ "coloca uno dentro" para usted). El destructor realiza la "limpieza final" en cada objeto de la clase, antes de que el sistema reclame la memoria de dicho objeto. Los destructores no pueden tomar argumentos y tampoco pueden sobrecargarse. En el capítulo 17, explicaremos con mayor detalle los constructores y los destructores.

Observe que las funciones que la clase proporciona al mundo exterior son precedidas por el especificador de acceso a miembros **`public`**. Las funciones **`public`** implementan comportamientos o servicios que la clase proporciona a sus clientes, por lo general llamadas *interfaz* de la clase o *interfaz pública*.



### Observación de ingeniería de software 16.3

*Los clientes tienen acceso a la interfaz de la clase, pero no deben tener acceso a la implementación de la clase.*

La definición de la clase contiene las declaraciones de los datos y de las funciones miembro. Las declaraciones de las funciones miembro son los prototipos de las funciones que explicamos en capítulos anteriores. Las funciones miembro pueden definirse dentro de una clase, pero es una buena práctica de programación definir dichas funciones fuera de la definición de la clase.



### Observación de ingeniería de software 16.4

*Declarar funciones miembro dentro de la definición de una clase (mediante sus prototipos de función), y definir dichas funciones miembro fuera de la definición de la clase, separa la interfaz de una clase de su implementación. Esto promueve la buena ingeniería de software. Los clientes de una clase no pueden ver la implementación de las funciones miembro de la clase y no necesitan recompilarlas si cambia la implementación.*

Observe el uso del *operador binario de resolución de alcance* (**`::`**) en cada definición de función miembro que se encuentra después de la definición de la clase, en la figura 16.2. Una vez que se define la clase y se declaran sus funciones miembro, éstas deben definirse. Cada función miembro de una clase puede ser definida directamente en el cuerpo de la clase (en lugar de incluir el prototipo de la función de la clase) o después del cuerpo de la clase. Cuando una función miembro es definida después de su correspondiente definición de clase, el nombre de la función es precedido por el nombre de la clase y el operador binario de resolución de alcance (**`::`**). Esto "une" el nombre del miembro con el nombre de la clase para identificar de manera única a las funciones de una clase en particular.

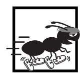


### Error común de programación 16.4

*Cuando se definen las funciones miembro de una clase fuera de ésta, es un error omitir el nombre de la clase y el operador de resolución de alcance en el nombre de la función.*

Aunque una función miembro declarada en la definición de una clase también puede definirse fuera de dicha definición, esa función miembro aún se encuentra dentro del *alcance de la clase*, es decir, su nombre solamente es conocido para otras funciones miembro de la clase, a menos que se haga referencia a éste mediante un objeto de la clase, mediante una referencia a un objeto de la clase o mediante un apuntador a un objeto de la clase. En un momento veremos más sobre el alcance de una clase.

Si la función miembro se define dentro de la definición de la clase, el compilador inserta el código de la función miembro. Las funciones miembro que se definen fuera de la definición de la clase pueden insertarse mediante la palabra reservada **`inline`**. Recuerde que el compilador se reserva el derecho de insertar o no el código de una función.



### Tip de rendimiento 16.1

*Definir una función miembro pequeña en la definición de la clase permite la inserción del código de ésta (si el compilador elige hacerlo). Esto puede mejorar el rendimiento, pero no promueve la mejor ingeniería de software, ya que los clientes de la clase podrán ver la implementación de la función y su código debe recompilarse, si la definición de función* ***`inline`*** *cambia.*

### Observación de ingeniería de software 16.5

*Solamente deben definirse dentro del encabezado de la clase las funciones miembro más sencillas y más estables (es decir, aquellas en las que es poco probable que ocurra un cambio).*

Es interesante que las funciones miembro **`imprimeMilitar`** e **`imprimeEstandar`** no toman argumentos. Esto se debe a que estas funciones miembro saben de manera implícita que imprimirán los datos miembro del objeto **`Hora`** particular para las que fueron invocadas. Esto hace que las llamadas a las funciones miembro sean más concisas que las llamadas a las funciones convencionales en la programación por procedimientos.



### Tip para prevenir errores 16.1

*El hecho de que las llamadas a funciones miembro por lo general no toman argumentos o toman menos argumentos que las llamadas a funciones convencionales de los lenguajes de programación no orientados a objetos, reduce la posibilidad de pasar argumentos erróneos, de tipo incorrecto o un número incorrecto de argumentos.*



### Observación de ingeniería de software 16.6

*A menudo, utilizar el método de la programación orientada a objetos simplifica las llamadas a funciones mediante la reducción del número de parámetros que se pasan. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de los datos y las funciones miembro dentro de un objeto permite que las funciones miembro tengan acceso a los datos miembro.*

Las clases simplifican la programación debido a que el cliente (o el usuario del objeto de la clase) solamente necesita preocuparse por las operaciones encapsuladas o incrustadas en el objeto. Por lo general, dichas operaciones se diseñan para estar orientadas al cliente, en lugar de estar orientadas a la implementación. Los clientes no necesitan preocuparse por la implementación de la clase (aunque, por supuesto, el cliente desea una implementación correcta y eficiente). Las interfaces cambian, pero con menos frecuencia que las implementaciones. Cuando una implementación cambia, el código que depende de la implementación debe cambiar en concordancia. Ocultar la implementación elimina la posibilidad de que otras partes del programa se hagan dependientes de los detalles de la implementación de la clase.

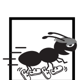


### Observación de ingeniería de software 16.7

*Uno de los temas centrales de este libro es "reutilizar, reutilizar, reutilizar". Explicaremos cuidadosamente un buen número de técnicas para "pulir" las clases y mejorar su uso. Nos enfocaremos en la "elaboración de clases útiles" y en la creación de "activos de software" útiles.*

A menudo, las clases no tienen que crearse "a partir de cero". En vez de ello, pueden incluir objetos de otras clases como miembros, o pueden *derivarse* a partir de otras clases que proporcionan atributos y comportamientos que las nuevas clases pueden utilizar. Tal *reutilización de software* puede mejorar en gran medida la productividad del programador. A la derivación de nuevas clases a partir de clases existentes se le llama *herencia*, y es el tema del capítulo 19. A la inclusión de objetos de clases como miembros de otras clases se le llama *composición* (o *agregación*) y la explicaremos en el capítulo 17.

La gente que es nueva en la programación orientada a objetos a menudo muestra preocupación por el hecho de que los objetos pudieran ser muy grandes debido a que contienen datos y funciones. Por lógica, esto es verdad; el programador podría pensar en los objetos como contenedores de datos y funciones. Sin embargo, físicamente esto no es verdad.



### Tip de rendimiento 16.2

*Los objetos sólo contienen datos, por lo que son mucho más pequeños que si además contuvieran funciones. Al aplicar el operador **`sizeof`** al nombre de una clase o a un objeto de dicha clase, éste reportará sólo el tamaño de los datos de dicha clase. El compilador crea una copia (solamente) de las funciones miembro, separada de todos los objetos de la clase. Todos los objetos de la clase comparten esta única copia de las funciones miembro. Por supuesto, cada objeto necesita su propia copia de los datos de la clase, ya que estos datos pueden variar entre los objetos. No se puede modificar el código de la función (también denominado código entrante o procedimiento puro) y, por lo tanto, se puede compartir entre todos los objetos de una clase.*

## 16.3 Alcance de una clase y acceso a los miembros de una clase

Los datos miembro de una clase (variables declaradas dentro de la definición de la clase) y las funciones miembro (funciones declaradas dentro de la definición de la clase) pertenecen al *alcance de esa clase*. Las funciones que no son miembros se definen con *alcance de archivo*.

Dentro del alcance de una clase se puede acceder de inmediato a los miembros de esa clase desde todas las funciones miembro de ésta, y se puede hacer referencia a ella por su nombre. Fuera del alcance de una clase, se hace referencia a los miembros de la clase a través de uno de los manipuladores de objeto: el nombre de un objeto, una referencia o un apuntador a un objeto. [En el capítulo 17, veremos que el compilador inserta un manipulador implícito en cada referencia a un dato miembro o función miembro de un objeto.]

Las funciones miembro de una clase pueden sobrecargarse, pero sólo mediante otras funciones miembro de dicha clase. Para sobrecargar una función miembro, simplemente proporcione un prototipo para cada versión de la función sobrecargada dentro de la definición de la clase, y proporcione una definición de función separada para cada versión de la función.

Las variables declaradas en la función miembro tienen un *alcance de función*; éstas se conocen solamente en dicha función. Si una función miembro define una variable con el mismo nombre que una variable con alcance de clase, la variable de alcance de clase se oculta detrás de la variable de alcance de archivo dentro de dicha función. Es posible acceder a dicha variable oculta, si antes del nombre de la función se coloca el nombre de la clase, seguido por el operador de resolución de alcance (**::**). Es posible acceder a las variables globales ocultas mediante el operador unario de resolución de alcance (vea el capítulo 15).

Los operadores que se utilizan para acceder a los miembros de una clase son idénticos a los operadores que se utilizan para acceder a los miembros de una estructura. El *operador punto de selección de miembros (.)* se combina con el nombre de un objeto o con la referencia a un objeto para acceder a los miembros de dicho objeto. El operador *flecha de selección de miembros (**->**)* se combina con un apuntador a un objeto para acceder a los miembros de dicho objeto.

La figura 16.3 utiliza una clase sencilla llamada **Cuenta** con el dato miembro público **x** de tipo **int** y la función miembro pública **imprime**, para ilustrar el acceso a los miembros de una clase mediante los operadores de selección de miembros. El programa define tres variables relacionadas con el tipo **Cuenta**: **contador**, **refContador** (la referencia a un objeto **Cuenta**) y **ptrContador** (un apuntador a un objeto **Cuenta**). La variable **refContador** hace referencia a **contador**, y la variable **ptrContador** apunta a **contador**. *Aquí es importante notar que el dato miembro* **x** *se declara como* **public** *solamente para mostrar la forma en que se accede a los miembros* **public** *sin manipuladores (es decir, un nombre, una referencia o un apuntador). Como establecimos, por lo general los datos se hacen* **private**, *tal como lo haremos en la mayoría de los ejemplos subsecuentes*. En el capítulo 19, en algunas ocasiones haremos que los datos sean **protected** (protegidos).

```
// Figura 16.3: fig16_03.cpp
// Demostración de los operadores de acceso a los miembros de una clase . y ->
//
// PRECAUCIÓN: EN EJEMPLOS POSTERIORES EVITAMOS LOS DATOS PUBLIC!
#include <iostream>

using std::cout;
using std::endl;

// Una clase sencilla Cuenta
class Cuenta {
public:
   int x;
   void imprime() { cout << x << endl; }
}; // fin de la clase Cuenta

int main()
{
   Cuenta contador,                 // crea el objeto contador
```

**Figura 16.3** Acceso a los datos y funciones miembro de un objeto a través de cada tipo de manipulador de objeto: nombre del objeto, una referencia al objeto y un apuntador al objeto. (Parte 1 de 2.)

```
          *ptrContador = &contador,  // apuntador hacia contador
          &refContador = contador;   // referencia hacia contador

   cout << "Asigna 7 a x y lo imprime utilizando el nombre del objeto: ";
   contador.x = 7;         // asigna 7 al dato miembro x
   contador.imprime();     // llama a la función miembro imprime

   cout << "Asigna 8 a x y lo imprime utilizando una referencia: ";
   refContador.x = 8;      // asigna 8 al dato miembro x
   refContador.imprime();  // llama a la función miembro imprime

   cout << "Asigna 10 a x y lo imprime utilizando un apuntador: ";
   ptrContador->x = 10;    // asigna 10 al dato miembro x
   ptrContador->imprime(); // llama a la función miembro imprime
   return 0;
} // fin de la función main
```

```
Asigna 7 a x y lo imprime utilizando el nombre del objeto: 7
Asigna 8 a x y lo imprime utilizando una referencia: 8
Asigna 10 a x y lo imprime utilizando un apuntador: 10
```

**Figura 16.3** Acceso a los datos y funciones miembro de un objeto a través de cada tipo de manipulador de objeto: nombre del objeto, una referencia al objeto y un apuntador al objeto. (Parte 2 de 2.)

## 16.4 Separación de la interfaz y la implementación

Uno de los principios fundamentales de la buena ingeniería de software es separar la interfaz de la implementación. Esto facilita la modificación de los programas. En lo que respecta a los clientes, los cambios en la implementación de la clase no lo afectan, mientras la interfaz original de la clase proporcionada al cliente permanezca sin cambios (la funcionalidad de la clase puede extenderse más allá de la interfaz original).



**Observación de ingeniería de software 16.8**

*Coloque la declaración de la clase en un archivo de encabezado para que cualquier cliente que desee utilizar la clase pueda incluirla. Esto conforma la interfaz pública de la clase (y proporciona al cliente los prototipos de las funciones que necesita para poder llamar a las funciones miembro de la clase). Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la implementación de la clase.*



**Observación de ingeniería de software 16.9**

*Los clientes de una clase no necesitan acceder al código fuente de la clase para poder utilizarla. Sin embargo, necesitan poder ligarse al código del objeto de la clase (es decir, a la versión compilada de la clase). Esto motiva a los fabricantes independientes de software a proporcionar bibliotecas de clases para su venta o en licencia los fabricantes independientes proporcionan en sus productos sólo archivos de encabezado y módulos de objetos. No se revela información alguna del propietario; lo que sí sucedería si se proporcionara el código fuente. La comunidad de usuarios de C++ se beneficia al tener disponibles más bibliotecas de clases producidas por proveedores.*

En realidad, las cosas no son tan sencillas. Los archivos de encabezado contienen algunas partes de la implementación y algunas pistas con respecto a otras. Por ejemplo, las funciones miembro **`inline`** deben estar en un archivo de encabezado, de manera que cuando el compilador compile un cliente, éste pueda incluir la definición de la función **`inline`** en su lugar. Los miembros privados de una clase se listan dentro de la definición de la clase en el archivo de encabezado, de manera que estos miembros son visibles a los clientes aún cuando éstos no acceden a los miembros privados.



**Observación de ingeniería de software 16.10**

*Es necesario incluir en el archivo de encabezado la información importante para la interfaz de una clase. La información que se utilizará sólo de manera interna dentro de la clase, y que no será necesaria para los clientes de la clase, debe incluirse en el archivo fuente no publicado. Éste es otro ejemplo del principio del menor privilegio.*

La figura 16.4 divide el programa de la figura 16.2 en múltiples archivos. Cuando se construye un programa en C++, por lo general cada definición de clase se coloca en un *archivo de encabezado*, y esas definiciones de las funciones miembro de la clase se colocan en un *archivo de código fuente* con el mismo nombre base (por convención). Los archivos de encabezado se incluyen (mediante **`#include`**) en cada uno de los archivos que utiliza la clase, y el archivo de código fuente se compila y se enlaza con el archivo que contiene el programa principal. Revise la documentación de su compilador para determinar cómo compilar y vincular programas que consisten en varios códigos fuente.

El programa consiste en el archivo de encabezado **`hora1.h`**, en el que se define la clase **`Hora`**, el archivo fuente **`hora1.cpp`**, en el que se definen las funciones miembro de la clase **`Hora`** y el código fuente **`fig16_04.cpp`** en el que se define la función **`main`**. La salida de este programa es idéntica a la salida de la figura 16.2.

```
// Figura 16.4: hora1.h
// Declaración de la clase Hora.
// Las funciones miembro están definidas en hora1.cpp

// evita inclusiones múltiples del archivo de encabezado
#ifndef HORA1_H
#define HORA1_H

// Definición del tipo de dato abstracto Hora
class Hora {
public:
   Hora();                              // constructor
   void estableceHora( int, int, int ); // establece hora, minuto, segundo
   void imprimeMilitar();               // imprime la hora en formato
                                           militar
   void imprimeEstandar();              // imprime la hora en formato
                                           estándar
private:
   int hora;      // 0 - 23
   int minuto;    // 0 - 59
   int segundo;   // 0 - 59
}; // fin de la clase Hora

#endif
```

**Figura 16.4** Separación de la interfaz y la implementación de la clase **`Hora`**; **`hora1.h`**.

```
// Figura 16.4: hora1.cpp
// Definiciones de las funciones miembro de la clase Hora.
#include <iostream>

using std::cout;

#include "hora1.h"

// El constructor Hora inicializa en cero a cada dato miembro.
// Garantiza que todos los objetos de Hora inician en un estado consistente.
Hora::Hora() { hora = minuto = segundo = 0; }

// Establece un nuevo valor de Hora por medio de la hora militar. Realiza
   verificaciones
```

**Figura 16.4** Separación de la interfaz y la implementación de la clase **`Hora`**; **`hora1.cpp`**. (Parte 1 de 2.)

```
// de validación de los valores de los datos. Establece en cero a los
   valores inválidos.
void Hora::estableceHora( int h, int m, int s )
{
   hora = ( h >= 0 && h < 24 ) ? h : 0;
   minuto = ( m >= 0 && m < 60 ) ? m : 0;
   segundo = ( s >= 0 && s < 60 ) ? s : 0;
} // fin de la función estableceHora

// Imprime Hora en formato militar
void Hora::imprimeMilitar()
{
   cout << ( hora < 10 ? “0” : “” ) << hora << “:”
        << ( minuto < 10 ? “0” : “” ) << minuto;
} // fin de la función imprimeMilitar

// Imprime Hora en formato estándar
void Hora::imprimeEstandar()
{
   cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
        << “:” << ( minuto < 10 ? “0” : “” ) << minuto
        << “:” << ( segundo < 10 ? “0” : “” ) << segundo
        << ( hora < 12 ? “ AM” : “ PM” );
} // fin de la función imprimeEstandar
```

**Figura 16.4** Separación de la interfaz y la implementación de la clase **Hora**; **hora1.cpp**. (Parte 2 de 2.)

```
// Figura 16.4: fig16_04.cpp
// Controlador para la clase hora1
// NOTA: Compílelo con hora1.cpp
#include <iostream>

using std::cout;
using std::endl;

#include “hora1.h”

// Controlador para probar la clase simple Hora
int main()
{
   Hora h;  // instancia el objeto h de la clase Hora

   cout << “La hora militar inicial es “;
   h.imprimeMilitar();
   cout << “\nLa hora estandar inicial es “;
   h.imprimeEstandar();

   h.estableceHora( 13, 27, 6 );
   cout << “\n\nLa hora militar despues de estableceHora es “;
   h.imprimeMilitar();
   cout << “\nLa hora estandar despues de estableceHora es “;
   h.imprimeEstandar();

```

**Figura 16.4** Separación de la interfaz y la implementación de la clase Hora; **fig16_04.cpp**. (Parte 1 de 2.)

```
    h.estableceHora( 99, 99, 99 );  // intenta establecer valores inválidos
    cout << “\n\nDespues de intentar establecer valores invalidos:n”
         << “\nHora militar: “;
    h.imprimeMilitar();
    cout << “\nHora estandar: “;
    h.imprimeEstandar();
    cout << endl;
    return 0;
} // fin de la función main
```

```
La hora militar inicial es 00:00
La hora estandar inicial es 12:00:00 AM

La hora militar despues de estableceHora es 13:27
La hora estandar despues de estableceHora es 1:27:06 PM

Despues de intentar establecer valores invalidos:
Hora militar: 00:00
Hora estandar: 12:00:00 AM
```

**Figura 16.4** Separación de la interfaz y la implementación de la clase Hora; **`fig16_04.cpp`**. (Parte 2 de 2.)

Observe que la declaración de la clase se encierra dentro del siguiente código de preprocesador:

```
// evita inclusiones múltiples del archivo de encabezado
#ifndef HORA1_H
#define HORA1_H
    ...
#endif
```

Cuando escribimos programas más grandes, también se colocan otras definiciones y declaraciones dentro de los archivos de encabezado. Las directivas de preprocesador anteriores evitan que se incluya el código que se encuentra entre la directiva **`#ifndef`** (que significa “si no está definido”) y la directiva **`#endif`**, si el nombre **`HORA1_H`** ya está definido. Si el encabezado no se incluyó antes dentro de un archivo, el nombre **`HORA1_H`** es definido por la directiva **`#define`**, y las instrucciones del archivo de encabezado se incluyen. Si el archivo de encabezado se incluyó previamente, entonces **`HORA1_H`** ya está definido, y el archivo de encabezado ya no se incluye de nuevo. Por lo general, los intentos de incluir un archivo de encabezado varias veces (de manera inadvertida) por lo general ocurren en programas grandes con muchos archivos de encabezado que podrían incluir otros archivos de encabezado. [*Nota:* La convención que utilizamos para el nombre de la constante simbólica dentro de las directivas del proprocesador es simplemente el nombre del archivo de encabezado con el guión bajo en lugar del punto.]

**Tip para prevenir errores 16.2**

*Utilice las directivas de preprocesador **`#ifndef`**, **`#define`** y **`#endif`**, para evitar que los archivos de encabezado se incluyan más de una vez en un programa.*



**Buena práctica de programación 16.2**

*Utilice el nombre del archivo de encabezado con un guión bajo, en lugar del punto dentro de las directivas de preprocesador **`#ifndef`** y **`#define`** de un archivo de encabezado.*

## 16.5 Control de acceso a miembros

Los especificadores de acceso a miembros **`public`** y **`private`** (y **`protected`**, como veremos en el capítulo 19) controlan el acceso a los datos y a las funciones miembro de una clase. El modo de acceso predeterminado para las clases es **`private`**, de modo que todos los miembros que se encuentran después del encabezado y antes del primer especificador de acceso a miembros son privados. Después de cada especificador de acceso a

miembros, se aplica el modo que llamó dicho especificador de acceso a miembros, hasta el siguiente especificador de acceso a miembros, o hasta la llave derecha de terminación (**}**) de la definición de la clase. Es posible repetir los especificadores de acceso a miembros **public**, **private** y **protected**, pero hacerlo no es común y puede resultar confuso.

Sólo se puede acceder a los miembros privados de una clase mediante funciones miembro (y funciones amigas, como veremos en el capítulo 17) de dicha clase. Es posible acceder a los miembros públicos de una clase a través de cualquier función dentro del programa.

El principal propósito de los miembros públicos es el de presentar a los clientes de una clase una vista de los *servicios* (comportamiento) que proporciona la clase. Este conjunto de servicios forma la *interfaz pública* de la clase. Los clientes de la clase no necesitan preocuparse por la forma en que la clase lleva a cabo sus tareas. Los miembros privados de una clase, así como las definiciones de sus funciones miembro públicas, no están accesibles para los clientes de la clase. Estos componentes forman la *implementación* de la clase.

### Observación de ingeniería de software 16.11

*C++ promueve que los programas sean independientes de la implementación. Cuando se modifica la implementación de una clase utilizada por código independiente de la implementación, dicho código no necesita modificarse. Si cambia cualquier parte de la interfaz de la clase, debe recompilarse el código independiente de la implementación.*



### Error común de programación 16.5

*El intento por parte de una función, que no es miembro de una clase en particular (o una amiga de esa clase), para acceder a los miembros privados de esa clase, es un error de sintaxis.*

La figura 16.5 demuestra que los miembros privados de la clase sólo están accesibles a través de la interfaz de la clase pública por medio de las funciones miembro públicas. Cuando este programa se compila, el compilador genera dos errores que establecen que el miembro privado especificado en cada instrucción no está accesible. La figura 16.5 incluye **hora1.h**, y se compila con **hora1.cpp** de la figura 16.4.



### Buena práctica de programación 16.3

*Si usted elige listar primero los miembros privados en la definición de la clase, utilice explícitamente el especificador de acceso a miembros* **private***, a pesar de que éste se asume de manera predeterminada. Esto mejora la claridad del programa.*

```
// Figura 16.5: fig16_05.cpp
// Demuestra los errores resultantes por intentar
// accede a los miembros privados de una clase.
#include <iostream>

using std::cout;

#include "hora1.h"

int main()
{
   Hora h;

   // Error: 'Hora::hora' no está accesible
   h.hora = 7;

   // Error: 'Hora::minuto' no está accesible
   cout << "minuto = " << h.minuto;

   return 0;
} // fin de la función main
```

**Figura 16.5** Intento erróneo para acceder a los miembros privados de una clase. (Parte 1 de 2.)

*Mensajes de error del compilador de Microsoft Visual C++*

```
Compiling...
fig16_05.ccp
C:\fig16_05.cpp(15) : error C2248: 'hora' : cannot access private member
declared in class 'Hora'
C:\fig16_05\ hora1.h(17) : see declaration of 'hora'
C:\fig16_05.cpp(18) : error C2248: 'minuto' : cannot access private member
declared in class 'Hora'
C:\hora1.h(18) : see declaration of 'minuto'
Error executing cl.exe.

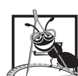
fig16_05.exe - 2 error(s), 0 warning(s)
```

**Figura 16.5** Intento erróneo para acceder a los miembros privados de una clase. (Parte 2 de 2.)



### Buena práctica de programación 16.4

*A pesar de que los especificadores de acceso a miembros* **`public`** *y* **`private`** *pueden repetirse e intercalarse, coloque primero todos los miembros públicos de una clase en un grupo, y después coloque todos los miembros privados en otro grupo. Esto centra la atención del cliente en la interfaz pública, en lugar de hacerlo en la implementación de la clase.*



### Observación de ingeniería de software 16.12

*Mantenga todos los datos de una clase como privados. Proporcione funciones miembro públicas para establecer los valores de los datos miembro privados y para obtener los valores de los datos miembro privados. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce errores y mejora la capacidad de modificación del programa.*

Un cliente de una clase puede ser una función miembro de otra clase, o puede ser una función global (es decir, una función estilo C de "pérdida" o de "liberación" dentro del archivo, tal como **`main`**, que no es una función miembro de ninguna clase).

El acceso predeterminado a los miembros de una clase es privado. El acceso a los miembros de una clase puede establecerse explícitamente como **`public`**, **`protected`** (como veremos en el capítulo 19) o **`private`**. El acceso predeterminado para los miembros de **`struct`** es **`public`**. El acceso a los miembros de **`struct`** también puede establecerse explícitamente como **`public`**, **`protected`**, o **`private`**, y se establece de manera predeterminada a **`public`**.



### Observación de ingeniería de software 16.13

*Los diseñadores de clases utilizan miembros* **`private`**, **`protected`** *y* **`public`** *para reforzar el concepto de ocultamiento de información y el del principio del menor privilegio.*

Sólo porque un dato de la clase sea privado no necesariamente significa que los clientes no puedan efectuar modificaciones a dichos datos. Los datos pueden modificarse mediante funciones miembro, a través de amigas de dicha clase. Como veremos, estas funciones deben estar diseñadas para garantizar la integridad de los datos.

El acceso a los datos privados debe controlarse cuidadosamente mediante las funciones miembro, llamadas *funciones de acceso* (también denominadas *métodos de acceso*). Por ejemplo, para permitir a los clientes leer el valor de datos privados, la clase proporciona una función *obtener (get)*. Para permitir a los clientes modificar datos privados, la clase puede proporcionar una función *establecer (set)*. Dicha modificación parecería violar la idea de los datos privados, pero una función miembro *establecer* puede proporcionar capacidades de validación (tales como verificación de rangos), para asegurarse de que el valor se estableció de manera correcta. Además, una función *establecer* puede traducir la forma de los datos utilizados en la interfaz a la forma utilizada en la implementación. Una función *obtener* no necesita mostrar los datos en formato "original"; en vez de ello, puede editar los datos y limitar la vista de los datos que el cliente verá.



### Observación de ingeniería de software 16.14

*El diseñador de la clase no necesita proporcionar funciones* obtener *o* establecer *para cada elemento privado de datos; estas capacidades solamente deben proporcionarse cuando sea apropiado. Si un servicio es útil para el código cliente, dicho servicio debe proporcionarse en la interfaz pública de la clase.*

**Tip para prevenir errores 16.3**

*Hacer que los datos miembro de una clase sean privados y que las funciones miembro de la clase sean públicas facilitan la corrección de errores, debido a que los problemas con la manipulación de datos se ubican en las funciones miembro de la clase o en las amigas de la clase.*

## 16.6 Funciones de acceso y funciones de utilidad

No todas las funciones miembro necesitan ser públicas para servir como parte de la interfaz de una clase. Algunas funciones miembro permanecen como privadas y sirven como *funciones de utilidad* para otras funciones de la clase.



**Observación de ingeniería de software 16.15**

*Las funciones miembro tienden a caer en ciertas categorías diferentes: funciones que leen y devuelven el valor de datos miembros privados; funciones que establecen el valor de datos miembros privados; funciones que implementan los servicios de la clase; y funciones que realizan distintas tareas mecánicas para la clase, tales como la inicialización de los objetos de una clase, la asignación de objetos de una clase, la conversión entre clases y tipos predefinidos o entre clases y otras clases, y la manipulación de memoria para los objetos de la clase.*

Las funciones de acceso a datos pueden leer o desplegar datos. Otro uso común para las funciones de acceso es la de comprobar la veracidad o falsedad de condiciones, dichas funciones a menudo se denominan *funciones predicado*. Un ejemplo de una función predicado es la función **`estaVacia`** para cualquier clase contenedora, es decir, para una clase capaz de almacenar muchos objetos, tales como una lista ligada, una pila o una cola. Un programa probaría **`estaVacia`** antes de intentar leer otro elemento desde el objeto contenedor. Una función predicado **`estaLlena`** podría evaluar un objeto de clase contenedora para determinar si la clase ya no tiene espacio libre. Las funciones predicado para nuestra clase **`Hora`** podrían ser **`esAM`** y **`esPM`**.

El programa que muestra la figura 16.6 muestra la idea de una *función de utilidad* (también llamada *función de ayuda*). Una función de utilidad no es parte de una interfaz pública de la clase, en vez de ello, es una

```
// Figura 16.6: vendedor.h
// Definición de la clase Vendedor
// Las funciones miembro están definidas en vendedor.cpp
#ifndef VENDEDOR_H
#define VENDEDOR_H

class Vendedor {
public:
   Vendedor();                         // constructor
   void obtieneVentasDelUsuario();     // obtiene cifras de ventas desde
                                          el teclado
   void estableceVentas( int, double ); // El usuario proporciona las cifras
                                       // de ventas de un mes.
   void imprimeVentasAnuales();

private:
   double totalVentasAnuales();        // función de utilidad
   double ventas[ 12 ];                // cifras de ventas de 12 meses
}; // fin de la clase Vendedor

#endif
```

**Figura 16.6** Uso de una función de utilidad; **`vendedor.h`**.

```
// Figura 16.6: vendedor.cpp
// Funciones miembro para la clase Vendedor
```

**Figura 16.6** Uso de una función de utilidad; **`vendedor.cpp`**. (Parte 1 de 3.)

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setprecision;
using std::setiosflags;
using std::ios;

#include “vendedor.h”

// La función constructor inicializa el arreglo
Vendedor::Vendedor()
{
   for ( int i = 0; i < 12; i++ )
      ventas[ i ] = 0.0;
} // fin del constructor Vendedor

// Función para obtener 12 cifras de ventas del usuario
// desde el teclado
void Vendedor::obtieneVentasDelUsuario()
{
   double montoVentas;

   for ( int i = 1; i <= 12; i++ ) {
      cout << “Introduzca el monto de las ventas de un mes “ << i << “: “;

      cin >> montoVentas;
      estableceVentas( i, montoVentas );
   } // fin de for
} // fin de la función obtieneVentasDelUsuario

// Función para establecer una de 12 cifras de ventas mensuales.
// Observe que el valor del mes debe ser de 0 a 11.
void Vendedor::estableceVentas( int mes, double monto )
{
   if ( mes >= 1 && mes <= 12 && monto > 0 )
      ventas[ mes - 1 ] = monto; // ajusta los subíndices 0-11
   else
      cout << “Mes o monto de ventas no valido” << endl;
} // fin de la función estableceVentas

// Imprime el total de las ventas anuales
void Vendedor::imprimeVentasAnuales()
{
   cout << setprecision( 2 )
        << setiosflags( ios::fixed | ios::showpoint )
        << “\nEl total de las ventas anuales es: $”
        << totalVentasAnuales() << endl;
} // fin de la función imprimeVentasAnuales

// Función de utilidad privada para totalizar las ventas anuales
```

**Figura 16.6** Uso de una función de utilidad; **`vendedor.cpp`**. (Parte 2 de 3.)

```
double Vendedor::totalVentasAnuales()
{
   double total = 0.0;

   for ( int i = 0; i < 12; i++ )
      total += ventas[ i ];

   return total;
} // fin de la función totalVentasAnuales
```

**Figura 16.6** Uso de una función de utilidad; **`vendedor.cpp`**. (Parte 3 de 3.)

```
// Figura 16.6: fig16_06.cpp
// Demostración de una función de utilidad
// Compílelo con vendedor.cpp
#include “vendedor.h”

int main()
{
   Vendedor v;                  // crea el objeto v de Vendedor

   v.obtieneVentasDelUsuario(); // observe el código secuencial simple
   v.imprimeVentasAnuales();    // no hay estructuras de control en main
   return 0;
} // fin de la función main
```

```
Introduzca el monto de las ventas de un mes 1: 5314.76
Introduzca el monto de las ventas de un mes 2: 4292.38
Introduzca el monto de las ventas de un mes 3: 4589.83
Introduzca el monto de las ventas de un mes 4: 5534.03
Introduzca el monto de las ventas de un mes 5: 4376.34
Introduzca el monto de las ventas de un mes 6: 5698.45
Introduzca el monto de las ventas de un mes 7: 4439.22
Introduzca el monto de las ventas de un mes 8: 5893.57
Introduzca el monto de las ventas de un mes 9: 4909.67
Introduzca el monto de las ventas de un mes 10: 5123.45
Introduzca el monto de las ventas de un mes 11: 4024.97
Introduzca el monto de las ventas de un mes 12: 5923.92

El total de las ventas anuales es: $60120.59
```

**Figura 16.6** Uso de una función de utilidad; **`fig16_06.cpp`**.

función miembro privadas que permite la operación de las funciones miembro públicas de la clase. La idea de las funciones de utilidad no es que las utilicen los clientes de una clase.

La clase **`Vendedor`** contiene un arreglo de 12 cifras de ventas mensuales a las cuales un constructor inicializa en cero y la función **`estableceVentas`** les asigna el valor definido por el usuario. La función miembro **`imprimeVentasAnuales`** imprime el total de ventas de los 12 meses anteriores. La función de utilidad **`totalVentasAnuales`** contiene el total de las cantidades vendidas de los últimos 12 meses para beneficio de **`imprimeVentasAnuales`**. La función miembro **`imprimeVentasAnuales`** edita las cantidades de ventas en formato de moneda.

Observe que **`main`** incluye solamente una secuencia simple de llamadas a las funciones miembro (no existen estructuras de control).



**Observación de ingeniería de software 16.16**

*Un fenómeno de la programación orientada a objetos es que una vez que se define una clase, por lo general la creación y la multiplicación de objetos de dicha clase implica solamente una sencilla secuencia de llamadas a funciones miembro; pocas, o ninguna estructura de control es necesaria. Por el contrario, es común tener estructuras de control en la implementación de las funciones miembro de una clase.*

## 16.7 Inicialización de los objetos de una clase: Constructores

Cuando se crea un objeto de una clase, sus miembros pueden inicializarse mediante una función *constructor* de dicha clase. Un constructor es una función miembro especial que tiene el mismo nombre que la clase y no devuelve un tipo de dato. El programador proporciona el constructor, el cual se invoca cada vez que se crea un objeto de dicha clase (se crea la instancia). Los constructores pueden sobrecargarse para producir distintas maneras de inicializar a los objetos de una clase. Los datos miembro pueden inicializarse dentro del constructor de una clase, o sus valores pueden establecerse posteriormente después de la creación del objeto. Sin embargo, es una buena práctica de ingeniería de software asegurarse de que un objeto se inicializa por completo antes de que el código cliente invoque a las funciones miembro del objeto. En general, no debe confiar en el código cliente para asegurarse de que un objeto se inicialice de manera correcta.



**Error común de programación 16.6**

*Los datos miembro de una clase no pueden inicializarse dentro de su definición.*



**Error común de programación 16.7**

*Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor, son errores de sintaxis.*



**Buena práctica de programación 16.5**

*Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicialice de manera apropiada con valores significativos. En especial, los datos miembro apuntadores deben inicializarse con un valor legítimo de apuntador, o con 0.*



**Tip para prevenir errores 16.4**

*Toda función miembro (y amiga) que modifique los datos miembro privados de un objeto debe garantizar que los datos restantes se encuentren en un estado consistente.*

Cuando se declara un objeto de la clase, es posible proporcionar *inicializadores* entre paréntesis a la derecha del nombre del objeto y antes del punto y coma. Estos inicializadores se pasan como argumentos al constructor de la clase. Pronto veremos diversos ejemplos sobre estas *llamadas a los constructores*. [*Nota*: aunque por lo general los programadores no llaman a los constructores, pueden proporcionar datos que se pasan a los constructores como argumentos.]

## 16.8 Uso de argumentos predeterminados con constructores

El constructor de **`hora1.cpp`** (figura 16.4) inicializa **`hora`**, **`minuto`** y **`segundo`** en 0 (es decir, 12 de la noche en horario militar). Los constructores pueden contener argumentos predeterminados. La figura 16.7 redefine la función constructor **`Hora`** para incluir argumentos predeterminados en cero para cada variable. Al proporcionar argumentos predeterminados al constructor, incluso si no se proporcionan valores en la llamada al constructor, se garantiza la inicialización del objeto a un estado consistente, debido a los argumentos predeterminados. Un constructor proporcionado por el programador que predetermina todos sus argumentos (o que no requiere argumentos explícitos) es también un *constructor predeterminado*, es decir, un constructor que se puede invocar sin argumentos. Solamente puede existir un constructor predeterminado por clase.

```
// Figura 16.7: hora2.h
// Declaración de la clase Hora.
// Las funciones miembro están definidas en hora2.cpp
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **`hora2.h`**. (Parte 1 de 2.)

```

// directivas de preprocesador que
// evitan inclusiones múltiples del archivo de encabezado
#ifndef HORA2_H
#define HORA2_H

// Definición del tipo de dato abstracto Hora
class Hora {
public:
   Hora( int = 0, int = 0, int = 0 );   // constructor predeterminado
   void estableceHora( int, int, int ); // establece hora, minuto, segundo
   void imrpimeMilitar();               // imprime la hora en formato
                                           militar
   void imrpimeEstandar();              // imprime la hora en formato
                                           estándar
private:
   int hora;      // 0 - 23
   int minuto;    // 0 - 59
   int segundo;   // 0 - 59
}; // fin de la clase Hora

#endif
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **`hora2.h`**. (Parte 2 de 2.)

```
// Figura 16.7: hora2.cpp
// Definiciones de las funciones miembro para la clase Hora.
#include <iostream>

using std::cout;

#include “hora2.h”

// El constructor Hora inicializa en cero a cada dato miembro.
// Garantiza que todos los objetos de Hora inician en un estado consistente.
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }

// Establece un nuevo valor de Hora, utilizando la hora militar. Realiza
   verificaciones de
// validez sobre los valores de datos. Establece en cero los valores no
   válidos.
void Hora::estableceHora ( int h, int m, int s )
{
   hora    = ( h >= 0 && h < 24 ) ? h : 0;
   minuto  = ( m >= 0 && m < 60 ) ? m : 0;
   segundo = ( s >= 0 && s < 60 ) ? s : 0;
} // fin de la función estableceHora

// Imprime Hora en formato militar
void Hora::imprimeMilitar()
{
   cout << ( hora < 10 ? “0” : “” ) << hora << “:”
        << ( minuto < 10 ? “0” : “” ) << minuto;
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **`hora2.cpp`**. (Parte 1 de 2.)

```
} // fin de la función imprimeMilitar

// Imprime Hora en formato estándar
void Hora::imprimeEstandar()
{
   cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
        << “:” << ( minuto < 10 ? “0” : “” ) << minuto
        << “:” << ( segundo < 10 ? “0” : “” ) << segundo
        << ( hora < 12 ? “ AM” : “ PM” );
} // fin de la función imprimeEstandar
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **`hora2.cpp`**. (Parte 2 de 2.)

```
// Figura 16.7: fig16_07.cpp
// Demostración de un constructor predeterminado
// función para la clase Hora.
#include <iostream>

using std::cout;
using std::endl;

#include “hora2.h”

int main()
{
   Hora h1,             // todos los argumentos predeterminados
        h2(2),          // minuto y segundo predeterminados
        h3(21, 34),     // segundo predeterminado
        h4(12, 25, 42), // todos los valores especificados
        h5(27, 74, 99); // todos los malos valores especificados

   cout << “Construida con:\n”
        << “todos los argumentos predeterminados:\n   “;
   h1.imprimeMilitar();
   cout << “\n   “;
   h1.imprimeEstandar();

   cout << “\nhora especificada; minuto y segundo predeterminados:”
        << “\n   “;
   h2.imprimeMilitar();
   cout << “\n   “;
   h2.imprimeEstandar();

   cout << “\nhora y minuto especificados; segundo predeterminado:”
        << “\n   “;
   h3.imprimeMilitar();
   cout << “\n   “;
   h3.imprimeEstandar();

   cout << “\nhora, minuto y segundo especificados:”
        << “\n   “;
   h4.imprimeMilitar();
   cout << “\n   “;
   h4.imprimeEstandar();
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **`fig16_07.cpp`**. (Parte 1 de 2.)

```

   cout << "\ntodos los valores no validos especificados:"
        << "\n   ";
   h5.imprimeMilitar();
   cout << "\n   ";
   h5.imprimeEstandar();
   cout << endl;

   return 0;
} // fin de la función main
```

```
Construida con:
todos los argumentos predeterminados:
   00:00
   12:00:00 AM
hora especificada; minuto y segundo predeterminados:
   02:00
   2:00:00 AM
hora y minuto especificados; segundo predeterminado:
   21:34
   9:34:00 PM
hora, minuto y segundo especificados:
   12:25
   12:25:42 PM
todos los valores no validos especificados:
   00:00
   12:00:00 AM
```

**Figura 16.7** Uso de un constructor con argumentos predeterminados; **fig16_07.cpp**. (Parte 2 de 2.)

En este programa, el constructor llama a la función **estableceHora** con los valores pasados al constructor (o con los valores predeterminados), para garantizar que el valor suministrado para hora se encuentra en el rango de 0 a 23, y que los valores para minuto y segundo se encuentran en el rango de 0 a 59. Si un valor se encuentra fuera de rango, dicho valor se establece en cero mediante **estableceHora** (para garantizar que cada dato miembro permanezca en estado consistente).

Observe que el constructor **Hora** podría escribirse para que incluyera las mismas instrucciones que la función miembro **estableceHora**. Esto podría ser un poco más eficiente debido a que se eliminaría la llamada adicional a **estableceHora**. Sin embargo, colocar el código del constructor **Hora** y la función miembro **estableceHora** de manera idéntica haría más difícil el mantenimiento de este programa. Si la implementación de la función miembro **estableceHora** cambia, la implementación del constructor **Hora** tiene que cambiar en concordancia. Hacer que el constructor **Hora** llame a **estableceHora** de manera directa requiere que cualquier cambio que se haga a la implementación de **estableceHora** se haga una sola vez. Esto reduce la probabilidad de errores cuando se altera la implementación. Además, el rendimiento del constructor **Hora** puede mejorarse si se declara el constructor explícitamente **inline**, o mediante la definición del constructor en la definición de la clase (la cual introduce implícitamente la definición de la función).

**Observación de ingeniería de software 16.17**

*Si la función miembro de una clase proporciona toda o parte de la funcionalidad requerida por el constructor (o alguna otra función miembro) de la clase, llame a dicha función miembro desde el constructor (u otra función miembro). Esto simplifica el mantenimiento del código y reduce la posibilidad de un error si se modifica la implementación del código. Como regla general, evite repetir el código.*



**Buena práctica de programación 16.6**

*Sólo declare argumentos predeterminados en el prototipo de la función dentro de la definición de la clase, en el archivo de encabezado.*



**Error común de programación 16.8**

*Especificar inicializadores predeterminados para la misma función miembro tanto en el encabezado como en la definición de la función miembro, es un error.*

[*Nota:* Cualquier cambio a los argumentos predeterminados de un método requiere que se recompile el código cliente. Si es probable que los valores predeterminados de los argumentos se modifiquen, mejor utilice funciones sobrecargadas. Así, si cambia la implementación de una función miembro, no se tendrá que recompilar el código cliente.]

La figura 16.7 inicializa cinco objetos de la clase **`Hora`**, uno con los tres argumentos predeterminados en la llamada al constructor, otro con un argumento especificado, otro con dos argumentos especificados, otro más con tres argumentos especificados y el último con tres argumentos no válidos especificados. El contenido de cada dato miembro, después de crear la instancia y realizar la inicialización del objeto, se despliega.

Si no se define un constructor para la clase, el compilador crea un constructor predeterminado. Dicho constructor no realiza inicialización alguna, de modo que cuando se crea el objeto, no existe la garantía de que se encuentre en un estado consistente.



**Observación de ingeniería de software 16.18**

*Es posible que una clase no contenga un constructor predeterminado, si cualquiera de los constructores está definido y ninguno de ellos es explícitamente un constructor predeterminado.*

## 16.9 Uso de destructores

Un *destructor* es otro tipo de función miembro especial de una clase. El nombre del destructor de una clase es el carácter *tilde*(~) seguido por el nombre de la clase. Esta convención es intuitivamente atractiva debido a que, como veremos en un capítulo posterior, el operador tilde es el operador de complemento a nivel de bits y, en cierto sentido, el destructor es el complemento del constructor.

Al destructor de una clase se le llama cuando se destruye un objeto. Esto ocurre cuando, por ejemplo, un objeto automático se destruye si la ejecución del programa rebasa el alcance en el que ese objeto fue creado. El destructor mismo no destruye realmente al objeto, éste realiza la *limpieza final* antes de que el sistema se lo pida a la memoria del objeto, para que ésta pueda reutilizarse para almacenar nuevos objetos.

Un destructor no recibe parámetros y no devuelve valor alguno. Una clase solamente puede tener un destructor; la sobrecarga de destructores no esta permitida.



**Error común de programación 16.9**

*Intentar pasar argumentos a un destructor para especificar un tipo de retorno para un destructor, para devolver valores de un destructor, o para sobrecargar un destructor, es un error de sintaxis (incluso* **`void`** *no puede especificarse).*

Observe que aun cuando no proporcionamos los destructores para las clases presentadas hasta el momento, toda clase tiene un destructor. Si el programador no proporciona explícitamente un destructor, el compilador crea un destructor "vacío". En el capítulo 18, construiremos destructores adecuados para las clases cuyos objetos contengan memoria asignada dinámicamente (por ejemplo, para arreglos o cadenas), o que utilizan otros recursos del sistema (por ejemplo, archivos de disco). En el capítulo 17, explicaremos cómo asignar y liberar memoria.



**Observación de ingeniería de software 16.19**

*Como veremos en lo que resta del libro, los constructores y los destructores son mucho más importantes en C++ y en la programación orientada a objetos, de lo que es posible dar a conocer después de la breve introducción que aquí presentamos.*

## 16.10 Invocación de constructores y destructores

Los constructores y los destructores son llamados automáticamente. El orden en el que ocurren estas llamadas a función depende del orden en el que la ejecución introduce y rebasa el alcance en el que estos objetos se crean. Por lo general, las llamadas a un destructor se hacen en orden inverso de las llamadas a un constructor. Sin em-

bargo, como veremos en la figura 16.8, las clases de almacenamiento de los objetos pueden alterar el orden en el que se llama a los destructores.

Los constructores son llamados por objetos definidos con alcance global, antes de que cualquier otra función (incluso **`main`**) en este archivo comience su ejecución (aunque el orden de la ejecución de constructores de objetos globales entre archivos no está garantizado). Los destructores correspondientes son llamados cuando termina **`main`**, o cuando se llama a la función **`exit`** (vea el capítulo 14). Los destructores no son llamados por objetos globales, si el programa termina con una llamada a una función **`exit`** o a una **`abort`** (vea el capítulo 14).

Se llama al constructor de un objeto local automático cuando la ejecución alcanza el punto en donde se definen los objetos. Los destructores correspondientes se llaman cuando los objetos salen de alcance (es decir, cuando se abandona el bloque en el que se definieron). Los constructores y los destructores de objetos automáticos se llaman cada vez que los objetos entran o salen de alcance. Los destructores de objetos automáticos no se llaman si el programa termina con una llamada a las funciones **`exit`** o **`abort`**.

Se llama al constructor para un objeto local estático solamente una vez cuando la ejecución alcanza por primera vez el punto donde el objeto está definido. Los destructores correspondientes se llaman cuando termina **`main`** cuando se llama a la función **`exit`**. No se llama a los destructores para objetos estáticos, si el programa termina con una llamada a una función **`abort`**.

El programa de la figura 16.8 muestra el orden en el que los constructores y los destructores son llamados para los objetos de la clase **`CreaYDestruye`** en distintos alcances. El programa define a primero con alcance global. Se llama a su constructor al comenzar la ejecución del programa y se llama a su destructor al terminar el programa, después de que los demás objetos son destruidos.

```
// Figura 16.8: crea.h
// Definición de la clase CreaYDestruye.
// Las funciones miembro están definidas en crea.cpp.
#ifndef CREA_H
#define CREA_H

class CreaYDestruye {
public:
   CreaYDestruye( int );  // constructor
   ~CreaYDestruye();      // destructor
private:
   int datos;
}; // fin de la clase CreaYDestruye

#endif
```

**Figura 16.8** Demostración del orden en el cual se llama a los constructores y a los destructores; **`crea.h`**.

```
// Figura 16.8: crea.cpp
// Definiciones de las funciones miembro para la clase CreaYDestruye
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

#include “crea.h”

```

**Figura 16.8** Demostración del orden en el cual se llama a los constructores y a los destructores; **`crea.cpp`**. (Parte 1 de 2.)

```
CreaYDestruye::CreaYDestruye( int valor )
{
   datos = valor;
   cout << “Objeto “ << datos << “   constructor”;
} // fin del constructor CreaYDestruye

CreaYDestruye::~CreaYDestruye()
   { cerr << “Objeto “ << datos << “   destructor “ << endl; }
```

**Figura 16.8** Demostración del orden en el cual se llama a los constructores y a los destructores; **crea.cpp**. (Parte 2 de 2.)

```
// Figura 16.8: fig16_08.cpp
// Demostración del orden en el que se llama a los constructores
// y a los destructores.
#include <iostream>

using std::cout;
using std::endl;

#include “crea.h”

void crea( void );   // prototipo

CreaYDestruye primero( 1 );            // objeto global

int main()
{
   cout << “   (global creado antes de main)” << endl;

   CreaYDestruye segundo( 2 );         // objeto local
   cout << “   (local automatico en main)” << endl;

   static CreaYDestruye tercero( 3 );  // objeto local
   cout << “   (local estatico en main)” << endl;

   crea();  // llamada a función para crear objetos

   CreaYDestruye cuarto( 4 );          // objeto local
   cout << “   (local automatico en main)” << endl;
   return 0;
} // fin de la función main

// Función para crear objetos
void crea( void )
{
   CreaYDestruye quinto( 5 );
   cout << “   (local automatico en crea)” << endl;

   static CreaYDestruye sexto( 6 );
   cout << “   (local estatico en crea)” << endl;

   CreaYDestruye septimo( 7 );
```

**Figura 16.8** Demostración del orden en el cual se llama a los constructores y a los destructores; **fig16_08.cpp**. (Parte 1 de 2.)

```
    cout << "   (local automatico en crea)" << endl;
} // fin de la función crea
```

```
Objeto 1   constructor   (global creado antes de main)
Objeto 2   constructor   (local automatico en main)
Objeto 3   constructor   (local estatico en main)
Objeto 5   constructor   (local automatico en crea)
Objeto 6   constructor   (local estatico en crea)
Objeto 7   constructor   (local automatico en crea)
Objeto 7   destructor
Objeto 5   destructor
Objeto 4   constructor   (local automatico en main)
Objeto 4   destructor
Objeto 2   destructor
Objeto 6   destructor
Objeto 3   destructor
```

**Figura 16.8** Demostración del orden en el cual se llama a los constructores y a los destructores; **fig16_08.cpp**. (Parte 2 de 2.)

La función **main** declara tres objetos. Los objetos **segundo** y **cuarto** son objetos locales automáticos, y el objeto **tercero** es un objeto local estático. Los constructores para cada uno de ellos se llaman cuando la ejecución alcanza el punto en donde se declara cada objeto. Los destructores de los objetos **cuarto** y **segundo** se llaman en ese orden, cuando se alcanza el final de **main**. El objeto **tercero** es **static**; por lo tanto, existe hasta que el programa termina. El destructor para el objeto **tercero** se llama antes del destructor para **primero**, pero después de la destrucción de todos los demás objetos.

La función crea declara tres objetos; **quinto** y **septimo** son objetos locales automáticos, y **sexto** es un objeto local estático. Los destructores para los objetos **septimo** y **quinto** se llaman en ese orden cuando se llega al final de **crea**. El objeto **sexto** es estático, de modo que existe hasta que termina el programa. El destructor para **sexto** se llama antes que los destructores para **tercero** y **primero**, pero después de la destrucción de los demás objetos.

## 16.11 Uso de datos miembro y funciones miembro

Se puede acceder a los datos miembro **private** de la clase solamente a través de las funciones miembro (y amigas) de la clase. Una manipulación típica puede ser el ajuste de saldos de un banco (por ejemplo, un dato miembro **private** de la clase **cuentaBanco**) por medio de la función miembro **calculaInteres**.

Con frecuencia, las clases proporcionan funciones miembro **public** para permitir a los clientes de la clase *establecer* (es decir, escribir) u *obtener* (es decir, leer) los valores de los datos miembro **private**. Estas funciones no necesitan llamarse específicamente *establecer* u *obtener*, pero por lo general así se llaman. De manera más específica, una función miembro que *establece* el dato miembro **tasaInteres** podría llamarse **estableceTasaInteres**, y una función miembro que *obtiene* la **tasaInteres** podría llamarse **obtieneTasaInteres**. Las funciones *obtener* también son conocidas como funciones de "consulta".

Podría parecer que proporcionar tanto las capacidades *establecer* como *obtener* es prácticamente lo mismo que hacer públicos los datos miembro. Ésta es otra más de las sutilezas de C++ que hacen tan deseable al lenguaje para la ingeniería de software. Si un dato miembro es público, entonces cualquier función del programa puede leerlo o escribirlo a voluntad. Si un dato miembro es privado, una función pública *obtener* parecería permitir a otras funciones leer la información a voluntad. Sin embargo, la función *obtener* podría controlar el formato en el que la información se devuelve el cliente. Una función pública *establecer* podría examinar cuidadosamente (y muy probablemente lo haría) cualquier intento de modificar el valor de los datos miembro. Esto garantiza que el nuevo valor sea adecuado para ese elemento de datos, es decir, que el elemento de datos permaneciera en un estado consistente. Por ejemplo, intentar *establecer* el día del mes en 37 se rechazaría, intentar

*establecer* una cantidad numérica en un valor alfabético se rechazaría, intentar *establecer* la calificación de un examen en 185 (cuando el rango adecuado es de cero a 100) también se rechazaría, etcétera.



### Observación de ingeniería de software 16.20

*Hacer privados a los datos miembro y controlar el acceso, en especial el acceso de escritura, para dichos datos miembro a través de funciones miembro públicas, ayuda a garantizar la integridad de los datos.*



### Tip para prevenir errores 16.5

*Los beneficios de la integridad de los datos no son automáticos por haber hecho privados a los datos miembro; el programador debe proporcionar una verificación de validez adecuada. Sin embargo, C++ proporciona un marco de trabajo en el que los programadores pueden diseñar mejores programas de manera conveniente.*



### Buena práctica de programación 16.7

*Las funciones miembro que establecen los valores de los datos privados deben verificar que los nuevos valores sean adecuados; si no lo son, las funciones* establecer *deben poner a los datos miembro privados en el estado consistente adecuado.*

El cliente de una clase debe ser notificado cuando se intenta asignar un valor no válido a un dato miembro. Las funciones *establecer* de una clase con frecuencia se escriben para devolver valores que indiquen que se intentó asignar un dato no válido a un objeto de la clase. Esto permite a los clientes de la clase probar los valores de devolución de las funciones *establecer*, para determinar si el objeto que están manipulando es un objeto válido, y para hacer lo adecuado si el objeto no lo es.

El programa de la figura 16.9 amplía la clase **`Hora`** para que incluya las funciones *establecer* y *obtener* correspondientes a los datos miembro privados **`hora`**, **`minuto`**, y **`segundo`**. Las funciones *establecer* controlan estrictamente la asignación de los datos miembro. Cualquier intento por *establecer* algún dato miembro en un valor incorrecto ocasionará que al dato miembro se le asigne cero (lo que dejará el dato miembro en un estado inconsistente). Cada función *obtener* simplemente devuelve el valor adecuado del dato miembro. Primero, el programa utiliza las funciones *establecer* para *poner* valores válidos a los datos miembro **`private`** del objeto **`h`** de **`Hora`**, después utiliza las funciones *obtener* para recuperar los valores para la salida. A continuación, las funciones *establecer* intentan *poner* valores inválidos a los miembros **`hora`** y **`segundo`**, y un valor válido al miembro **`minuto`**; después, las funciones *obtener* recuperan los valores para la salida. La salida confirma que los valores inválidos provocan que los datos miembro se *establezcan* en cero. Por último, el programa *establece* la hora en 11:58:00, e incrementa el valor de minuto por 3 mediante la llamada a la función **`incrementaMinutos`**. La función **`incrementaMinutos`** es una función no miembro que utiliza las funciones miembro *obtener* y *establecer* para incrementar apropiadamente al miembro minuto. Aunque esto funciona, afecta al rendimiento al hacer llamadas múltiples a la función. En el siguiente capítulo explicaremos la noción de funciones amigas como medio para eliminar esta carga en el rendimiento.

```
// Figura 16.09: hora3.h
// Declaración de la clase Hora.
// Las funciones miembro están definidas en tiempo3.cpp

// directivas de preprocesador que
// evitan inclusiones múltiples del archivo de encabezado
#ifndef HORA3_H
#define HORA3_H

class Hora {
public:
   Hora( int = 0, int = 0, int = 0 );  // constructor

   // funciones establecer
   void estableceHora( int, int, int ); // establece hora, minuto, segundo
   void estableceHora( int );           // establece hora
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener*, **`hora3.h`**. (Parte 1 de 6.)

```
   void estableceMinuto( int );         // establece minuto
   void estableceSegundo( int );        // establece segundo

   // funciones obtener
   int obtieneHora();                   // devuelve hora
   int obtieneMinuto();                 // devuelve minuto
   int obtieneSegundo();                // devuelve segundo

   void imprimeMilitar();               // despliega la hora militar
   void imprimeEstandar();              // despliega la hora estándar

private:
   int hora;                            // 0 - 23
   int minuto;                          // 0 - 59
   int segundo;                         // 0 - 59
}; // fin de la clase Hora

#endif
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener;* **`hora3.h`**. (Parte 2 de 6.)

```
// Figura 16.09: hora3.cpp
// Definiciones de las funciones miembro de la clase Hora.
#include <iostream>

using std::cout;

#include “hora3.h”

// Función constructor para inicializar los datos privados.
// Llama a la función miembro estableceHora para establecer variables.
// Los valores predeterminados son 0 (vea la definición de la clase).
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }

// Establece los valores para hora, minuto y segundo.
void Hora::estableceHora( int h, int m, int s )
{
   estableceHora( h );
   estableceMinuto( m );
   estableceSegundo( s );
} // fin de la función estableceHora

// Establece el valor de hora
void Hora::estableceHora( int h )
   { hora = ( h >= 0 && h < 24 ) ? h : 0; }

// Establece el valor de minuto
void Hora::estableceMinuto( int m )
   { minuto = ( m >= 0 && m < 60 ) ? m : 0; }

// Establece el valor de segundo
void Hora::estableceSegundo( int s )
   { segundo = ( s >= 0 && s < 60 ) ? s : 0; }
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener;* **`hora3.cpp`**. (Parte 3 de 6.)

```

// Obtiene el valor de hora
int Hora::obtieneHora() { return hora; }

// Obtiene el valor de minuto
int Hora::obtieneMinuto() { return minuto; }

// Obtiene el valor de segundo
int Hora::obtieneSegundo() { return segundo; }

// Imprime la hora en formato militar
void Hora::imprimeMilitar()
{
   cout << ( hora < 10 ? “0” : “” ) << hora << “:”
        << ( minuto < 10 ? “0” : “” ) << minuto;
} // fin de la función imprimeMilitar

// Imprime la hora en formato estándar
void Hora::imprimeEstandar()
{
   cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
        << “:” << ( minuto < 10 ? “0” : “” ) << minuto
        << “:” << ( segundo < 10 ? “0” : “” ) << segundo
        << ( hora < 12 ? “ AM” : “ PM” );
} // fin de la función imprimeEstandar
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener;* **`hora3.cpp`**. (Parte 4 de 6.)

```
// Figura 16.09: fig16_09.cpp
// Demostración de las funciones establecer y obtener de la clase Hora
#include <iostream>

using std::cout;
using std::endl;

#include “hora3.h”

void incrementaMinutos( Hora &, const int );

int main()
{
   Hora h;

   h.estableceHora( 17 );
   h.estableceMinuto( 34 );
   h.estableceSegundo( 25 );

   cout << “Resultado de establecer todos los valores validos:\n”
        << “  Hora: “    << h.obtieneHora()
        << “  Minuto: “  << h.obtieneMinuto()
        << “  Segundo: “ << h.obtieneSegundo();

   h.estableceHora( 234 );    // una hora inválida se establece en 0
   h.estableceMinuto( 43 );
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener;* **`fig16_09.cpp`**. (Parte 5 de 6.)

```
   h.estableceSegundo( 6373 ); // un segundo inválido se establece en 0

   cout << "\n\nResultado de intentar establecer una hora y un segundo
        << " no validos:\n  Hora: " << h.obtieneHora()
        << "  Minuto: "  << h.obtieneMinuto()
        << "  Segundo: " << h.obtieneSegundo() << "\n\n";

   h.estableceHora( 11, 58, 0 );
   incrementaMinutos( h, 3 );

   return 0;
} // fin de la función main

void incrementaMinutos( Hora &hh, const int cuenta )
{
   cout << "Incrementando minuto " << cuenta
        << " veces:\nHora inicial: ";
   hh.imprimeEstandar();

   for ( int i = 0; i < cuenta; i++ ) {
        hh.estableceMinuto( ( hh.obtieneMinuto() + 1 ) % 60);

      if ( hh.obtieneMinuto() == 0 )
        hh.estableceHora( ( hh.obtieneHora() + 1 ) % 24);

      cout << "\nminuto + 1: ";
        hh.imprimeEstandar();
   } // fin de for

   cout << endl;
} // fin de la función incrementaMinutos
```

```
Resultado de establecer todos los valores validos:
  Hora: 17  Minuto: 34  Segundo: 25

Resultado de intentar establecer una hora y un segundo no validos:
  Hora: 0  Minuto: 43  Segundo: 0
```

```
Incrementando minuto 3 veces:
Hora inicial: 11:58:00 AM
minuto + 1: 11:59:00 AM
minuto + 1: 12:00:00 PM
minuto + 1: 12:01:00 PM
```

**Figura 16.9** Uso de las funciones *establecer* y *obtener;* **`fig16_09.cpp`**. (Parte 6 de 6.)



### Error común de programación 16.10

*Un constructor puede llamar a otras funciones miembro de la clase, como funciones* establecer *y* obtener, *pero debido a que el constructor inicializa al objeto, es posible que los datos miembro aún no se encuentren en un estado consistente. Utilizar datos miembro antes de que se hayan inicializado adecuadamente puede ocasionar errores lógicos.*

El uso de funciones *establecer* es muy importante desde un punto de vista de ingeniería de software, ya que éstas pueden realizar un análisis de validación. Tanto las funciones *establecer* como las funciones *obtener* tienen otra importante ventaja para la ingeniería de software.



**Observación de ingeniería de software 16.21**

*Acceder a datos privados a través de funciones miembro* establecer *y* obtener *no sólo protege a los datos miembro de recibir valores no válidos, sino también protege a los clientes de la clase de la representación de los datos miembro. Entonces, si la representación de los datos cambia por alguna razón (por lo general para reducir la cantidad de almacenamiento requerida o para mejorar el rendimiento), sólo las funciones miembro necesitan cambiar; los clientes no necesitarán cambio alguno, mientras la interfaz provista por las funciones miembro permanezca igual. Sin embargo, los clientes necesitarán recompilarse.*

## 16.12 Una trampa sutil: Retorno de una referencia a un dato miembro privado

Una referencia a un objeto es un alias para el *nombre* del objeto y, por lo tanto, puede utilizarse en el lado izquierdo de una instrucción de asignación. En este contexto, la referencia hace a un *lvalue* perfectamente aceptable para recibir un valor. Una forma de utilizar esta capacidad (¡por desgracia!) es para hacer que una función miembro pública de una clase devuelva una referencia no constante a un dato miembro privado de esa clase.

La figura 16.10 utiliza una clase simplificada **`Hora`** para mostrar el retorno de una referencia a un dato miembro privado. Dicho retorno en realidad hace de la llamada a la función miembro **`malEstablecimientoHora`**, ¡un alias de la función miembro **`private hora`**! La llamada a la función puede utilizarse en cualquier forma en la que se puede utilizar un dato miembro **`private`**, ¡incluso como un *lvalue* dentro de una instrucción de asignación!

```
// Figura 16.10: hora4.h
// Declaración de la clase Hora.
// Las funciones miembro están definidas en hora4.cpp

// directivas de preprocesador que
// evitan inclusiones múltiples del archivo de encabezado
#ifndef HORA4_H
#define HORA4_H

class Hora {
public:
   Hora( int = 0, int = 0, int = 0 );
   void estableceHora( int, int, int );
   int obtieneHora();
   int &malEstablecimientoHora( int );  // retorno de referencia PELIGROSO
private:
   int hora;
   int minuto;
   int segundo;
}; // fin de la clase Hora

#endif
```

**Figura 16.10** Retorno de una referencia a un dato miembro privado; **`hora4.h`**. (Parte 1 de 5.)

```
// Figura 16.10: hora4.cpp
// Definiciones de las funciones miembro para la clase Hora.
#include "hora4.h"

// Función constructor para inicializar datos privados.
// Llama a la función miembro estableceHora para establecer variables.
// Los valores predeterminados son 0 (vea la definición de la clase).
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }

```

**Figura 16.10** Retorno de una referencia a un dato miembro privado; **`hora4.ccp`**. (Parte 2 de 5.)

```
// Establece los valores de hora, minuto y segundo.
void Hora::estableceHora( int h, int m, int s )
{
   hora    = ( h >= 0 && h < 24 ) ? h : 0;
   minuto  = ( m >= 0 && m < 60 ) ? m : 0;
   segundo = ( s >= 0 && s < 60 ) ? s : 0;
} // fin de la función estableceHora

// Obtiene el valor de hora
int Hora::obtieneHora() { return hora; }

// MALA PRÁCTICA DE PROGRAMACIÓN:
// Devolver una referencia a un miembro privado.
int &Hora::malEstablecimientoHora( int mh )
{
   hora = ( mh >= 0 && mh < 24 ) ? mh : 0;

   return hora;  // retorno de referencia PELIGROSO
} // fin de la función malEstablecimientoHora
```

**Figura 16.10** Retorno de una referencia a un dato miembro privado; **`hora4.ccp`**. (Parte 3 de 5.)

```
// Figura 16.10: fig16_10.cpp
// Demostración de una función miembro que
// devuelve una referencia a un dato miembro privado.
// La clase Hora se simplificó para este ejemplo.
#include <iostream>

using std::cout;
using std::endl;

#include “hora4.h”

int main()
{
   Hora h;
   int &refHora = h.malEstablecimientoHora( 20 );

   cout << “Hora antes de la modificacion: “ << refHora;
   refHora = 30;  // modificación con un valor inválido
   cout << “\nHora despues de la modificacion: “ << h.obtieneHora();

   // Peligroso: Llamada a una función que devuelve
   // una referencia que puede utilizarse como un lvalue!
   h.malEstablecimientoHora( 12 ) = 74;
   cout << “\n\n*********************************\n”
        << “MALA PRACTICA DE PROGRAMACION!!!!!!!!\n”
        << “malEstablecimientoHora como un lvalue, Hora: “
        << h.obtieneHora()
        << “\n*********************************” << endl;

   return 0;
} // fin de la función main
```

**Figura 16.10** Retorno de una referencia a un dato miembro privado; **`fig16_10.cpp`**. (Parte 4 de 5.)

```
Hora antes de la modificacion: 20
Hora despues de la modificacion: 30

********************************
MALA PRACTICA DE PROGRAMACION!!!!!!!!
malEstablecimientoHora como un lvalue, Hora: 74
********************************
```

**Figura 16.10** Retorno de una referencia a un dato miembro privado; **`fig16_10.cpp`**. (Parte 5 de 5.)



**Buena práctica de programación 16.8**

*Nunca haga que una función miembro pública devuelva una referencia no constante (o apuntador) a un dato miembro privado. Devolver una referencia como ésa viola el encapsulamiento de la clase. De hecho, devolver cualquier referencia a un apuntador a datos privados hace dependiente al código cliente, en cuanto a la representación de los datos de la clase. Entonces, devolver apuntadores o referencias a datos privados es una práctica peligrosa que debería evitarse.*

El programa comienza por declarar el objeto **`h`** de **`Hora`** y la referencia **`refHora`** al que se asigna la referencia devuelta por la llamada **`h.malEstablecimientoHora(20)`**. El programa despliega el valor del alias **`refHora`**. A continuación, el alias se utiliza para establecer el valor de la hora en 30 (un valor inválido), y el valor se despliega de nuevo. Por último, la llamada a la función por sí misma se utiliza como un *lvalue*, y se le asigna el valor 74 (otro valor inválido), y se despliega el valor.

## 16.13 Asignación mediante la copia predeterminada de miembros

El operador de asignación (**`=`**) puede utilizarse para asignar un objeto a otro objeto del mismo tipo. De manera predeterminada, tal asignación se lleva a cabo mediante la *copia de miembros*; cada miembro del objeto a la derecha del operador de asignación se copia (se asigna) de manera individual al mismo miembro en otro objeto (vea la figura 16.11). [*Nota:* La copia de miembros puede ocasionar serios problemas, cuando se utiliza con una clase cuyos datos miembro contienen apuntadores hacia memoria asignada dinámicamente; en el capítulo 18, explicaremos estos problemas y mostraremos cómo lidiar con ellos.]

```
// Figura 16.11: fig16_11.cpp
// Demostración de que los objetos de una clase pueden asignarse
// entre sí, por medio de una copia predeterminada de miembros
#include <iostream>

using std::cout;
using std::endl;

// Una clase simple Fecha
class Fecha {
public:
   Fecha( int = 1, int = 1, int = 1990 ); //  constructor predeterminado
   void imprime();
private:
   int mes;
   int dia;
   int anio;
}; // fin de la clase Fecha

// Constructor de la función simple Fecha sin verificación de rangos
```

**Figura 16.11** Asignación de un objeto a otro mediante la copia predeterminada de miembros. (Parte 1 de 2.)

```
Fecha::Fecha( int m, int d, int a )
{
   mes = m;
   dia = d;
   anio = a;
} // fin del constructor Fecha

// Imprime la Fecha en la forma mm-dd-aaaa
void Fecha::imprime()
   { cout << mes << '-' << dia << '-' << anio; }

int main()
{
   Fecha fecha1( 7, 4, 1993 ), fecha2;  // f2 da de manera predeterminada 1/1/90

   cout << "fecha1 = ";
   fecha1.imprime();
   cout << "\nfecha2 = ";
   fecha2.imprime();

   fecha2 = fecha1;   // asignación por la copia predeterminada de miembros
   cout << "\n\nDespues de la copia predeterminada de miembros, fecha2 = ";
   fecha2.imprime();
   cout << endl;

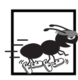
   return 0;
} // fin de la función main
```

```
fecha1 = 7-4-1993
fecha2 = 1-1-1990

Despues de la copia predeterminada de miembros, fecha2 = 7-4-1993
```

**Figura 16.11** Asignación de un objeto a otro mediante la copia predeterminada de miembros. (Parte 2 de 2.)

Los objetos pueden pasarse como argumentos de función y pueden devolverse desde funciones. Dicho paso y retorno se realiza mediante una llamada por valor predeterminada; se pasa o se devuelve una copia del objeto (presentaremos varios ejemplos en el capítulo 18).



**Tip de rendimiento 16.3**

*Pasar un objeto por valor es bueno desde el punto de vista de seguridad, ya que la función llamada no tiene acceso al objeto original de la función que llama, pero pasar por valor puede degradar el rendimiento, cuando se hace una copia de un objeto grande. Un objeto puede pasarse por referencia mediante el paso de un apuntador o de una referencia hacia el objeto. Pasar por referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, ya que a la función llamada se le da acceso al objeto original. Pasar por medio de una referencia constante es una alternativa segura y con buen rendimiento.*

## 16.14 Reutilización de software

La gente que escribe programas orientados a objetos se concentra en implementar clases útiles. Existe una gran oportunidad de capturar y catalogar clases para que puedan estar disponibles para grandes segmentos de la comunidad de programación. Existen muchas *bibliotecas de clases* importantes y otras que se están desarrollando alrededor del mundo. El software se construye cada vez más a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portables, de alto rendimiento y que están amplia-

mente disponibles. Esta clase de reutilización de software agiliza el desarrollo de software poderoso y de alta calidad. El *desarrollo rápido de aplicaciones* a través de los mecanismos de reutilización de componentes se ha convertido en un campo importante.

Sin embargo, los problemas importantes deben resolverse antes de que pueda llevarse a cabo una reutilización de software con todo su potencial. Necesitamos catalogar esquemas, mecanismos de protección para garantizar que las copias maestras de las clases no se corrompan, esquemas de descripción para que los diseñadores de nuevos sistemas puedan determinar qué clases están disponibles y qué tan cercanos se encuentran de cumplir con los requerimientos del desarrollador de software, etcétera. Muchos problemas de investigación y desarrollo interesantes deben resolverse. Existe una gran motivación para resolver estos problemas, ya que el valor potencial de sus soluciones es enorme.

## RESUMEN

- Las estructuras son tipos de datos adicionales construidas mediante el uso de otros tipos de datos.
- La palabra reservada **`struct`** introduce la definición de una estructura. El cuerpo de una estructura se delimita mediante llaves (**`{`** y **`}`**). Toda definición de una estructura debe terminar con punto y coma.
- El nombre de la etiqueta de una estructura puede utilizarse para declarar variables del tipo de esa estructura.
- Las definiciones de estructuras no reservan espacio en memoria; crean nuevos tipos de datos que se utilizan para declarar variables.
- Se accede a los miembros de una estructura o de una clase mediante los operadores de acceso a miembros, es decir, mediante el operador punto (**`.`**) y el operador flecha (**`->`**). El operador punto accede a los miembros de una estructura mediante el nombre de la variable o la referencia al objeto. El operador flecha accede a los miembros de una estructura mediante el apuntador al objeto.
- Las desventajas de crear nuevos tipos de datos mediante el elemento **`struct`** son la posibilidad de tener datos sin inicializar, e inicializaciones incorrectas; todos los programas que utilizan **`struct`** deben modificarse, si la implementación de **`struct`** cambia y si no se proporcionó protección alguna para garantizar que los datos se mantengan en estado consistente con los valores de datos apropiados.
- Las clases permiten al programador modelar objetos con atributos y comportamientos. Los tipos de clases de C++ pueden definirse con las palabras reservadas **`class`** y **`struct`**, sin embargo, por lo general se utiliza la palabra **`class`** para este propósito.
- El nombre de una clase puede utilizarse como un nombre de tipo para declarar objetos de dicha clase.
- Las definiciones de clases comienzan con la palabra reservada **`class`**. El cuerpo de la definición de la clase se delimita con las llaves (**`{`** y **`}`**). Las definiciones de clases terminan con punto y coma.
- Cualquier dato miembro o función miembro que se declara después de **`public:`** en una clase, es accesible a cualquier función con acceso a un objeto de la clase.
- Cualquier dato miembro o función miembro que se declara después de **`private:`** en una clase, es accesible sólo a amigas y otros miembros de la misma clase.
- Los especificadores de acceso a miembros siempre terminan con dos puntos (**`:`**), y pueden aparecer más de una vez en cualquier orden en la definición de la clase.
- Los datos privados no son accesibles desde afuera de la clase.
- La implementación de una clase debe ocultarse a sus clientes.
- Un constructor es una función miembro especial con el mismo nombre de la clase y sin valor de retorno; se utiliza para inicializar los miembros de objetos de dicha clase. Se llama al constructor de una clase cuado se crea la instancia de un objeto de esa clase.
- Una función que tiene el mismo nombre que su clase, pero que está precedida por el carácter tilde (**`~`**), se llama destructor.
- Al conjunto de funciones miembro **`public`** de una clase se le llama interfaz de una clase o interfaz pública.
- Cuando una función miembro se define fuera de la definición de la clase, el nombre de la función debe ser precedido por el nombre de la clase y por el operador binario de resolución de alcance (**`::`**).
- Las funciones miembro definidas mediante el operador unario de resolución de alcance fuera de la definición de una clase, se encuentra dentro del alcance de ésta.
- Las funciones miembro definidas en la definición de una clase se declaran de manera implícita como **`inline`**. El compilador se reserva el derecho de colocar o no cualquier función como **`inline`**.

- Invocar a funciones miembro es más conciso que llamar a funciones en la programación por procedimientos, debido a que se puede acceder a la mayoría de los datos utilizados por la función miembro dentro del objeto.
- Dentro del alcance de una clase se puede hacer referencia a sus miembros simplemente por su nombre. Fuera del alcance de la clase, se hace referencia a sus miembros a través del nombre del objeto, una referencia a un objeto o un apuntador a un objeto.
- Un principio fundamental de la buena ingeniería de software es separar la interfaz de la implementación.
- Por lo general, las definiciones de clases se colocan en archivos de encabezado y las definiciones de las funciones miembro se colocan dentro del código fuente de los archivos con el mismo nombre base.
- El modo predeterminado de acceso a clases es **`private`**, de modo que todos los miembros que se encuentran después del encabezado de la clase y antes del primer especificador de acceso a un miembro de la clase se consideran privados.
- Los miembros públicos de una clase presentan una vista de los servicios que proporciona la clase a sus clientes.
- El acceso a los datos **`private`** de una clase puede controlarse cuidadosamente mediante las funciones miembro llamadas funciones de acceso. Si una clase quiere permitir a sus clientes leer datos **`private`**, ésta puede proporcionar una función *obtener*. Para permitir a los clientes modificar los datos **`private`**, la clase puede proporcionar una función *establecer*.
- Por lo general, los datos miembro de una clase son de tipo **`private`**, y las funciones miembro de una clase son **`public`**. Algunas funciones miembro pueden ser privadas y servir como funciones de utilidad para las otras funciones de la clase.
- Los datos miembro de una clase no pueden inicializarse dentro de la definición de la clase. Éstos deben inicializarse dentro de un constructor o sus valores deben *establecerse* después de que su objeto fue creado.
- Los constructores pueden sobrecargarse.
- Una vez que se inicializa un objeto de la clase de manera apropiada, todas las funciones miembro que manipulan al objeto deben asegurarse de que el objeto permanece en un estado consistente.
- Cuando se declara un objeto de una clase, pueden proporcionarse inicializadores. Estos inicializadores se pasan al constructor de una clase.
- Los constructores pueden especificar argumentos predeterminados.
- Los constructores podrían no especificar tipos de retorno, ni intentar la devolución de valores.
- Si no se define un constructor para una clase, el compilador crea un constructor predeterminado. Un constructor predeterminado suministrado por el compilador no realiza inicialización alguna, por lo que cuando se crea un objeto de la clase, no se garantiza que el objeto se encuentre en un estado consistente.
- Se invoca al destructor de un objeto automático, cuando el objeto sale de su alcance (es decir, la ejecución deja el bloque en el cual se define el objeto). El destructor en sí mismo, en realidad no destruye al objeto, pero realiza la limpieza final antes de que el sistema reclame la memoria del objeto.
- Los destructores no reciben parámetros y no devuelven valores. Una clase solamente puede tener un destructor (los destructores no pueden sobrecargarse).
- El operador de asignación (**`=`**) se utiliza para asignar un objeto a otro objeto del mismo tipo. Por lo general, dicha asignación se realiza de manera predeterminada mediante la asignación de miembros. La asignación de miembros no es ideal para todas las clases.

## TERMINOLOGÍA

alcance de archivo
alcance de una clase
archivo de código fuente
archivo de encabezado
atributo
**`class`**
cliente de una clase
código reutilizable
comportamiento
constructor
constructor predeterminado
control de acceso a miembros
copia de miembros
crear la instancia de una clase (objeto)
dato miembro
definición de una clase
desarrollo rápido de aplicaciones
destructor
diseño orientado a objetos
encapsulamiento
especificadores de acceso a miembros
estado consistente de un dato miembro
estructura
extensibilidad
función de acceso
función de ayuda
función de consulta
función de utilidad
función *establecer*
función miembro
función miembro **`inline`**
función no miembro
función *obtener*
función predicado
implementación de una clase
inicializador de miembros

inicializar un objeto de una clase
instancia de una clase
interfaz de una clase
interfaz pública de una clase
mensaje
objeto
objeto global
objeto local estático
objeto local no estático
ocultamiento de información
operador binario de resolución de alcance (`::`)
operador de resolución de alcance (`::`)
operador de selección de miembros (`.` y `->`)
operador de selección de miembros de una clase (`.`)
operador flecha (`->`) de selección de miembros
operador punto (`.`) de selección de miembros
principio del menor privilegio
**`private`**
programación orientada a objetos (POO)
programación por procedimientos
**`protected`**
**`public`**
reutilización de software
servicios de una clase
tilde (`~`) en el nombre del destructor
tipo de dato
tipo de dato abstracto (ADT)
tipo definido por el programador
tipo definido por el usuario

## ERRORES COMUNES DE PROGRAMACIÓN

**16.1** Olvidar el punto y coma al final de una definición de clase (o de una estructura), es un error de sintaxis.

**16.2** Especificar un tipo o un valor de retorno para un constructor, es un error de sintaxis.

**16.3** Intentar inicializar explícitamente un dato miembro de una clase dentro de la definición de la clase, es un error de sintaxis.

**16.4** Cuando se definen las funciones miembro de una clase fuera de ésta, es un error omitir el nombre de la clase y el operador de resolución de alcance en el nombre de la función.

**16.5** El intento por parte de una función, que no es miembro de una clase en particular (o una amiga de esa clase), para acceder a los miembros privados de esa clase, es un error de sintaxis.

**16.6** Los datos miembro de una clase no pueden inicializarse dentro de su definición.

**16.7** Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor, son errores de sintaxis.

**16.8** Especificar inicializadores predeterminados para la misma función miembro tanto en el encabezado como en la definición de la función miembro, es un error.

**16.9** Intentar pasar argumentos a un destructor para especificar un tipo de retorno para un destructor, para devolver valores de un destructor, o para sobrecargar un destructor, es un error de sintaxis (incluso **`void`** no puede especificarse).

**16.10** Un constructor puede llamar a otras funciones miembro de la clase, como funciones *establecer* y *obtener*, pero debido a que el constructor inicializa al objeto, es posible que los datos miembro aún no se encuentren en un estado consistente. Utilizar datos miembro antes de que se hayan inicializado adecuadamente puede ocasionar errores lógicos.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**16.1** Para mayor claridad, utilice cada especificador de acceso a miembros una sola vez dentro de la definición de la clase. Primero coloque los elementos **`public`** en donde sean fáciles de localizar.

**16.2** Utilice el nombre del archivo de encabezado con un guión bajo en lugar del punto dentro de las directivas de preprocesador **`#ifndef`** y **`#define`** de un archivo de encabezado.

**16.3** Si usted elige listar primero los miembros privados en la definición de la clase, utilice explícitamente el especificador de acceso a miembros **`private`**, a pesar de que éste se asume de manera predeterminada. Esto mejora la claridad del programa.

**16.4** A pesar de que los especificadores de acceso a miembros **`public`** y **`private`** pueden repetirse e intercalarse, coloque primero todos los miembros públicos de una clase en un grupo, y después coloque todos los miembros privados en otro grupo. Esto centra la atención del cliente en la interfaz pública, en lugar de hacerlo en la implementación de la clase.

**16.5** Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicialice de manera apropiada con valores significativos. En especial, los datos miembro apuntadores deben inicializarse con un valor legítimo de apuntador, o con 0.

**16.6** Sólo declare argumentos predeterminados en el prototipo de la función dentro de la definición de la clase, en el archivo de encabezado.

**16.7** Las funciones miembro que establecen los valores de los datos privados deben verificar que los nuevos valores sean adecuados; si no lo son, las funciones *establecer* deben poner a los datos miembro privados en el estado consistente adecuado.

**16.8** Nunca haga que una función miembro pública devuelva una referencia no constante (o apuntador) a un dato miembro privado. Devolver una referencia como ésa viola el encapsulamiento de la clase. De hecho, devolver cualquier referencia a un apuntador a datos privados hace dependiente al código cliente, en cuanto a la representación de los datos de la clase. Entonces, devolver apuntadores o referencias a datos privados es una práctica peligrosa que debería evitarse.

## TIPS DE RENDIMIENTO

**16.1** Definir una función miembro pequeña en la definición de la clase permite la inserción del código de ésta (si el compilador elige hacerlo). Esto puede mejorar el rendimiento, pero no promueve la mejor ingeniería de software, ya que los clientes de la clase podrán ver la implementación de la función y su código debe recompilarse, si la definición de función **`inline`** cambia.

**16.2** Los objetos sólo contienen datos, por lo que son mucho más pequeños que sí además contuvieran funciones. Al aplicar el operador **`sizeof`** al nombre de una clase o a un objeto de dicha clase, éste reportará sólo el tamaño de los datos de dicha clase. El compilador crea una copia (solamente) de las funciones miembro, separada de todos los objetos de la clase. Todos los objetos de la clase comparten esta única copia de las funciones miembro. Por supuesto, cada objeto necesita su propia copia de los datos de la clase, ya que estos datos pueden variar entre los objetos. No se puede modificar el código de la función (también denominado código entrante o procedimiento puro) y, por lo tanto, se puede compartir entre todos los objetos de una clase.

**16.3** Pasar un objeto por valor es bueno desde el punto de vista de seguridad, ya que la función llamada no tiene acceso al objeto original de la función que llama, pero pasar por valor puede degradar el rendimiento, cuando se hace una copia de un objeto grande. Un objeto puede pasarse por referencia mediante el paso de un apuntador o de una referencia hacia el objeto. Pasar por referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, ya que a la función llamada se le da acceso al objeto original. Pasar por medio de una referencia constante es una alternativa segura y con buen rendimiento.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**16.1** Los clientes de una clase la utilizan sin conocer los detalles internos acerca de la manera en que se implementa. Si se modifica la implementación de una clase (por ejemplo, para mejorar el rendimiento), debido a que la interfaz de la clase permanece constante, el código fuente cliente de la clase no requiere modificación alguna (aunque el código cliente deberá compilarse de nuevo). Esto hace mucho más fácil la modificación de sistemas.

**16.2** Por lo general, las funciones miembro son más pequeñas que las que se encuentran en programas no orientados a objetos debido a que los datos almacenados en los datos miembro se validan por medio del constructor, o por medio de las funciones miembro que almacenan los nuevos datos. Debido a que los datos ya se encuentran en el objeto, las llamadas a las funciones miembro a menudo se hacen sin argumentos, o al menos tienen menos argumentos que las típicas llamadas a funciones en lenguajes no orientados a objetos. Por lo tanto, las llamadas, las definiciones de función y los prototipos de las funciones son más cortos.

**16.3** Los clientes tienen acceso a la interfaz de la clase, pero no deben tener acceso a la implementación de la clase.

**16.4** Declarar funciones miembro dentro de la definición de una clase (mediante sus prototipos de función) y definir dichas funciones miembro fuera de la definición de la clase separa la interfaz de una clase de su implementación. Esto promueve la buena ingeniería de software. Los clientes de una clase no pueden ver la implementación de las funciones miembro de la clase y no necesitan recompilarlos si cambia la implementación.

**16.5** Solamente deben definirse dentro del encabezado de la clase las funciones miembro más sencillas y más estables (es decir, aquellas en las que es poco probable que ocurra un cambio).

**16.6** A menudo, utilizar el método de la programación orientada a objetos simplifica las llamadas a funciones mediante la reducción del número de parámetros que se pasan. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de los datos y las funciones miembro dentro de un objeto permite que las funciones miembro tengan acceso a los datos miembro.

**16.7** Uno de los temas centrales de este libro es "reutilizar, reutilizar, reutilizar". Explicaremos cuidadosamente un buen número de técnicas para "pulir" las clases y mejorar su uso. Nos enfocaremos en la "elaboración de clases útiles" y en la creación de "activos de software" útiles.

**16.8** Coloque la declaración de la clase en un archivo de encabezado para que cualquier cliente que desee utilizar la clase pueda incluirla. Esto conforma la interfaz pública de la clase (y proporciona al cliente los prototipos de las funciones que necesita para poder llamar a las funciones miembro de la clase). Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la implementación de la clase.

**16.9** Los clientes de una clase no necesitan acceder al código fuente de la clase para poder utilizarla. Sin embargo, necesitan poder ligarse al código del objeto de la clase (es decir, a la versión compilada de la clase). Esto motiva a los fabricantes independientes de software a proporcionar bibliotecas de clases para su venta o en licencia. No se revela información alguna del propietario; lo que sí sucedería si se proporcionara el código fuente. La comunidad de usuarios de C++ se beneficia al tener disponibles más bibliotecas de clases producidas por proveedores.

**16.10** Es necesario incluir en el archivo de encabezado la información importante para la interfaz de una clase. La información que se utilizará sólo de manera interna dentro de la clase, y que no será necesaria para los clientes de la clase, debe incluirse en el archivo fuente no publicado. Éste es otro ejemplo del principio del menor privilegio.

**16.11** C++ promueve que los programas sean independientes de la implementación. Cuando se modifica la implementación de una clase utilizada por código independiente de la implementación, dicho código no necesita modificarse. Si cambia cualquier parte de la interfaz de la clase, debe recompilarse el código independiente de la implementación.

**16.12** Mantenga todos los datos de una clase como privados. Proporcione funciones miembro públicas para *establecer* los valores de los datos miembro privados y para *obtener* los valores de los datos miembro privados. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce errores y mejora la capacidad de modificación del programa.

**16.13** Los diseñadores de clases utilizan miembros **`private`**, **`protected`** y **`public`** para reforzar el concepto de ocultamiento de información y del principio del menor privilegio.

**16.14** El diseñador de la clase no necesita proporcionar funciones *obtener* o *establecer* para cada elemento privado de datos, estas capacidades solamente deben proporcionarse cuando sea apropiado. Si un servicio es útil para el código cliente, dicho servicio debe proporcionarse en la interfaz pública de la clase.

**16.15** Las funciones miembro tienden a caer en ciertas categorías diferentes: funciones que leen y devuelven el valor de datos miembros privados; funciones que establecen el valor de datos miembros privados; funciones que implementan los servicios de la clase; y funciones que realizan distintas tareas mecánicas para la clase, tales como la inicialización de los objetos de una clase, la asignación de objetos de una clase, la conversión entre clases y tipos predefinidos o entre clases y otras clases, y la manipulación de memoria para los objetos de la clase.

**16.16** Un fenómeno de la programación orientada a objetos es que una vez que se define una clase, por lo general la creación y la multiplicación de objetos de dicha clase implica solamente una sencilla secuencia de llamadas a funciones miembro; pocas, o ninguna estructura de control es necesaria. Por el contrario, es común tener estructuras de control en la implementación de las funciones miembro de una clase.

**16.17** Si la función miembro de una clase proporciona toda o parte de la funcionalidad requerida por el constructor (o alguna otra función miembro) de la clase, llame a dicha función miembro desde el constructor (u otra función miembro). Esto simplifica el mantenimiento del código y reduce la posibilidad de un error si se modifica la implementación del código. Como regla general, evite repetir el código.

**16.18** Es posible que una clase no contenga un constructor predeterminado, si cualquiera de los constructores está definido y ninguno de ellos es explícitamente un constructor predeterminado.

**16.19** Como veremos en lo que resta del libro, los constructores y los destructores son mucho más importantes en C++ y en la programación orientada a objetos, de lo que es posible dar a conocer después de la breve introducción que aquí presentamos.

**16.20** Hacer privados a los datos miembro y controlar el acceso, en especial el acceso de escritura, para dichos datos miembro a través de funciones miembro públicas, ayuda a garantizar la integridad de los datos.

**16.21** Acceder a datos privados a través de funciones miembro *establecer* y *obtener* no sólo protege a los datos miembro de recibir valores no válidos, sino también protege a los clientes de la clase de la representación de los datos miembro. Entonces, si la representación de los datos cambia por alguna razón (por lo general para reducir la cantidad de almacenamiento requerida o para mejorar el rendimiento), sólo las funciones miembro necesitan cambiar; los clientes no necesitarán cambio alguno, mientras la interfaz provista por las funciones miembro permanezca igual. Sin embargo, los clientes necesitarán recompilarse.

## *TIPS PARA PREVENIR ERRORES*

**16.1** El hecho de que las llamadas a funciones miembro por lo general no toman argumentos o toman menos argumentos que las llamadas a funciones convencionales de los lenguajes de programación no orientados a objetos, reduce la posibilidad de pasar argumentos erróneos, de tipo incorrecto o un número incorrecto de argumentos.

**16.2** Utilice las directivas de preprocesador **#ifndef**, **#define** y **#endif**, para prevenir que los archivos de encabezado se incluyan más de una vez en un programa.

**16.3** Hace que los datos miembro de una clase sean privados y que las funciones miembro de la clase sean públicas facilitan la corrección de errores debido a que los problemas con la manipulación de datos se ubican en las funciones miembro de la clase o en las amigas de la clase.

**16.4** Toda función miembro (y amiga) que modifique los datos miembros privados de un objeto debe garantizar que los datos restantes se encuentren en un estado consistente.

**16.5** Los beneficios de la integridad de los datos no son automáticos por haber hecho privados a los datos miembro; el programador debe proporcionar una verificación de validez adecuada. Sin embargo, C++ proporciona un marco de trabajo en el que los programadores pueden diseñar mejores programas de manera conveniente.

## EJERCICIOS DE AUTOEVALUACIÓN

**16.1** Complete los espacios en blanco:

a) Se accede a los miembros de una clase mediante el operador ____________ junto con el nombre de un objeto de la clase, o mediante el operador ____________ junto con un apuntador a un objeto de la clase.

b) Los miembros de una clase especificados como ____________ son accesibles a las funciones miembro de la clase y a las amigas de la clase.

c) Un ____________ es una función miembro especial utilizada para inicializar los datos miembro de una clase.

d) El acceso predeterminado para los miembros de una clase es ____________.

e) Una función ____________ se utiliza para asignar valores a datos miembro privados de una clase.

f) ____________ puede utilizarse para asignar un objeto de una clase a otro objeto de la misma clase.

g) Por lo general, las funciones miembro de una clase son ____________, y los datos miembro de una clase por lo general son ____________.

h) Una función ____________ se utiliza para recuperar valores de los datos privados de una clase.

i) Al conjunto de funciones miembro públicas de una clase se les llama ____________ de una clase.

j) Se dice que la implementación de una clase se oculta a sus clientes o que está ____________.

k) Las palabras reservadas ____________ y ____________ pueden utilizarse para introducir la definición de una clase.

l) Los miembros de una clase que se especifican como ____________ están accesibles en cualquier parte dentro del alcance del objeto de la clase.

**16.2** Encuentre los errores en cada uno de los siguientes segmentos de código, y explique cómo corregirlos:

a) Suponga que se declara el siguiente prototipo dentro de la clase **Hora**:

```
void ~Hora ( int );
```

b) La siguiente es una definición parcial de la clase **Hora**:

```
class Hora {
public:
// prototipos de la función
private:
   int hora = 0;
   int minuto = 0;
   int segundo = 0;
}; // fin de la clase Hora
```

c) Suponga que se declara el siguiente prototipo dentro de la clase **Empleado**:

```
int Empleado( const char *, const char * );
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**16.1** a) Punto (**.**), flecha (**->**). b) **private**. c) Constructor. d) **private**. e) *Establecer*. f) Copia predeterminada de miembros (realizada por el operador de asignación). g) **public**, **private**. h) *Obtener*. i) Interfaz. j) encapsulada. k) **class**, **struct**. l) **public**.

**16.2** a) Error: no se permite a los destructores devolver valores o tomar argumentos.
Corrección: elimine de la declaración el tipo de retorno **void** y el parámetro **int**.

b) Error: los miembros no pueden inicializarse de manera explícita en la definición de la clase.
   Corrección: elimine la inicialización explícita de la definición de la clase, e inicialice los datos miembro en un constructor.
c) Error: no se permite a los constructores devolver valores.
   Corrección: elimine el tipo de retorno **`int`** de la declaración.

## EJERCICIOS

**16.3** ¿Cuál es el propósito del operador de resolución de alcance?

**16.4** Proporcione un constructor que sea capaz de utilizar la hora actual de la función **`time()`**, declarada en la biblioteca estándar **`<ctime.h>`** de C++, para inicializar un objeto de la clase **`Hora`**.

**16.5** Cree una clase llamada **`Complejo`** para realizar aritmética con números complejos. Escriba un programa controlador para probar sus clases.

Los números complejos tienen la forma:

```
parteReal + parteImaginaria * i
```

en donde ***i*** es

$$\sqrt{-1}$$

Utilice variables **`double`** para representar datos de tipo **`private`** de una clase. Proporcione un constructor que permita inicializar un objeto de esta clase cuando se declare. El constructor debe contener valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione funciones miembro de tipo **`public`** para cada uno de los siguientes:

a) Suma de dos números complejos: las partes reales se suman juntas y las partes imaginarias se suman juntas.
b) Resta de dos números complejos: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Impresión de números complejos de la forma (**`a`**, **`b`**), en donde **`a`** es la parte real y **`b`** es la parte imaginaria.

**16.6** Cree una clase llamada **`Racional`** para realizar aritmética con fracciones. Escriba un programa controlador para evaluar su clase.

Utilice variables enteras para representar los datos de tipo **`private`** de la clase, es decir, el numerador y el denominador. Proporcione un constructor que permita a un objeto de esta clase que se inicialice cuando se declare. El constructor debe contener valores predeterminados, en caso de que no se proporcionen inicializadores, y debe almacenar la fracción en su forma reducida. Por ejemplo, la fracción:

$$\frac{2}{4}$$

se almacenaría en el objeto como **`1`** en el numerador y **`2`** en el denominador. Proporcione una función miembro **`public`** para realizar cada una de las siguientes tareas:

a) Suma de dos números racionales: el resultado debe almacenarse en forma reducida.
b) Resta de dos números racionales: el resultado debe almacenarse en forma reducida.
c) Multiplicación de dos números racionales: el resultado debe almacenarse en forma reducida.
d) División de dos números racionales: el resultado debe almacenarse en forma reducida.
e) Impresión de números racionales de la forma **`a/b`**, en donde **`a`** es el numerador y **`b`** es el denominador.
f) Impresión de números racionales en formato de punto flotante.

**16.7** Genere una clase **`Rectangulo`** con los atributos **`longitud`** y **`ancho`**, cada uno con un valor predeterminado igual a **`1`**. Proporcione funciones miembro que calculen el perímetro y el área del rectángulo. Además, proporcione las funciones *establecer* y obtener para los atributos **`longitud`** y **`ancho`**. Las funciones *establecer* deben verificar que **`longitud`** y **`ancho`** contengan números de punto flotante mayores que **`0.0`** y menores que **`20.0`**.

**16.8** Cree una clase **`Rectangulo`** más sofisticada que la que creó en el ejercicio 16.7. Esta clase sólo almacena las coordenadas cartesianas de las cuatro esquinas del rectángulo. El constructor llama a una función *establecer* que acepta cuatro coordenadas y verifica que cada una de éstas se encuentre en el primer cuadrante y que ninguna coordenada ***`x`*** o ***`y`*** sea mayor que **`20.0`**. La función *establecer* verifica también que las coordenadas proporcionadas formen en realidad un rectángulo. Proporcione funciones miembro que calculen la longitud, el ancho, el perímetro y el área. La longitud es la mayor de las dos dimensiones. Incluya una función predicado **`cuadrado`** que determine si el rectángulo es un cuadrado.

**16.9** Modifique la clase **`Rectangulo`** del ejercicio 16.8 para incluir una función **`dibujar`** que despliegue el rectángulo en una caja de 25 por 25 y que contenga el primer cuadrante en el cual reside el rectángulo. Incluya una función **`estableceCaracRelleno`** para especificar el carácter con el cual se rellenará el rectángulo. Incluya una función **`estableceCaracPerimetro`** para especificar que el carácter se utilizará para dibujar el borde del rectángulo, rotarlo, y moverlo alrededor dentro de la porción designada del primer cuadrante.

**16.10** Genere una clase **`EnteroMuyLargo`** que utilice un arreglo de 40 elementos para almacenar enteros hasta de 40 dígitos de longitud. Proporcione las funciones miembro **`entrada`**, **`salida`**, **`suma`** y **`resta`**. Para comprar objetos de **`EnteroMuyLargo`** proporcione las funciones **`esIgualQue`**, **`esDiferenteQue`**, **`esMayorQue`**, **`esMenorQue`**, **`esMayoroIgualQue`**, **`esMenoroIgualQue`**; cada una de éstas es una función "predicado" que simplemente devuelve **`verdadero`**, si se cumple la relación de los enteros muy largos, y **`falso`** si la relación no se cumple. Además, proporcione una función predicado **`esCero`**. Si se siente ambicioso, proporcione las funciones miembro **`multiplica`**, **`divide`** y **`modulo`**.

**16.11** Genere la clase **`Gato`** que le permita escribir un programa completo para jugar el juego del gato. La clase contiene como datos **`private`**, un arreglo de enteros **`double`** de 3 por 3. El constructor debe inicializar todo el tablero en cero. Permita dos jugadores humanos. A cualquier lugar donde el primer jugador mueva, coloque un 1 en el cuadro especificado; coloque un 2 en cualquier lugar en donde el segundo jugador haga un movimiento. Cada movimiento debe ser hacia un cuadro vacío. Después de cada movimiento, determine si alguien ganó el juego o si es un empate. Si se siente ambicioso, modifique su programa de manera que la computadora haga los movimientos para uno de los jugadores. Además, permita al jugador especificar si desea tirar primero o segundo. Si usted se siente particularmente ambicioso, desarrolle un programa que juegue un gato en tres dimensiones sobre un tablero de 4 por 4. (*Precaución:* éste es un proyecto sumamente desafiante que le podría tomar semanas de esfuerzo.)

# 17

# Clases en C++: Parte II

## Objetivos

- Crear y destruir objetos dinámicamente.
- Especificar objetos y funciones miembro **`const`** (constantes).
- Comprender el propósito de las funciones y las clases **`friend`** (amigas).
- Comprender cómo utilizar datos y funciones miembro **`static`**.
- Comprender el concepto de una clase contenedora.
- Comprender el concepto de clases iteradoras que recorren los elementos de clases contenedoras.
- Comprender el uso del apuntador **`this`**.




*Pero, para cumplir nuestros propios objetivos,*
*¿Olvidamos las burlas de nuestros amigos?*
Charles Churchill

*En lugar de esta absurda división de sexos, deberían clasificar*
*a la gente como estática y dinámica.*
Evelyn Waugh

*Por encima de todo: sé auténtico.*
William Shakespeare

*No tengas amigos diferentes a ti mismo.*
Confucio

## Plan general



## 17.1 Introducción

En este capítulo continuaremos nuestro estudio de las clases y la abstracción de datos. Explicaremos temas más avanzados y prepararemos el terreno para explicar las clases y la sobrecarga de operadores en el capítulo 18. La explicación en los capítulos 16 a 18 motiva a los programadores a utilizar objetos, lo que llamamos *programación basada en objetos (PBO)*. Luego, en los capítulos 19 y 20 presentamos la herencia y el polimorfismo; las técnicas de la verdadera *programación orientada a objetos (POO)*. En éste y en varios capítulos subsiguientes, utilizaremos las cadenas al estilo C que introdujimos en el capítulo 8. Esto lo ayudará a dominar el complejo tema de los apuntadores en C y a prepararse para el mundo profesional, en el cual verá una gran cantidad de código C heredado a lo largo de las dos últimas décadas.

## 17.2 Objetos y funciones miembro `const` (constantes)

Hemos puesto énfasis en el *principio del menor privilegio* como uno de los principios fundamentales de la buena ingeniería de software. Veamos ahora cómo se aplica este principio a los objetos.

Algunos objetos necesitan modificaciones y otros no. El programador puede utilizar la palabra **`const`** para especificar que un objeto no puede modificarse, y que cualquier intento por modificar el objeto es un error de sintaxis. Por ejemplo,

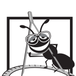
```
const Hora mediodia(12, 0, 0);
```

declara un objeto **`const`** de la clase **`Hora`** llamado **`mediodia`**, y lo inicializa a las 12 del medio día.



**Observación de ingeniería de software 17.1**

*Declarar un objeto como **`const`** ayuda a reforzar el principio del menor privilegio. Los intentos para modificar al objeto son captados en tiempo de compilación, en lugar de provocar errores en tiempo de ejecución.*



**Observación de ingeniería de software 17.2**

*Utilizar **`const`** es crucial para el diseño apropiado de clases y para la codificación y diseño de programas.*



**Tip de rendimiento 17.1**

*Declarar variables y objetos **`const`** no solamente es una práctica de ingeniería de software efectiva, también puede mejorar el rendimiento debido a que los sofisticados compiladores actuales pueden realizar ciertas optimizaciones sobre constantes, que no es posible realizar sobre variables.*

Los compiladores de C++ no permiten las llamadas de funciones miembro a objetos **const**, a menos que las funciones miembro por sí mismas también se declaren **const**. Esto es verdad incluso para las funciones miembro *obtener* que no modifican al objeto. Las funciones miembro declaradas **const** no pueden modificar al objeto, el compilador no permite esto.

Una función se especifica como **const** tanto en su prototipo como en su definición, al insertar la palabra reservada **const** después de la lista de parámetros de la función, y, en el caso de la definición de la función, antes de la llave izquierda que inicia el cuerpo de la función. Por ejemplo, la siguiente función miembro de la clase **A**

```
int A::obtieneValor() const { return datoMiembroPrivado; }
```

simplemente devuelve el valor de uno de los datos miembro del objeto, y se declara apropiadamente como **const**.



**Error común de programación 17.1**

*Definir como **const** una función miembro que modifica un dato miembro de un objeto, es un error de sintaxis.*



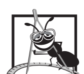
**Error común de programación 17.2**

*Definir como **const** una función miembro que llama a una función miembro no **const** de la clase en la misma instancia de la clase, es un error de sintaxis.*



**Error común de programación 17.3**

*Invocar a una función miembro no **const** en un objeto **const**, es un error de sintaxis.*



**Observación de ingeniería de software 17.3**

*Una función miembro **const** puede sobrecargarse con una versión no **const**. La elección respecto a cuál función miembro sobrecargada utilizar la hace el compilador, basándose en si el objeto es o no **const**.*

Aquí surge un problema interesante para los constructores y los destructores, que con frecuencia necesitan modificar objetos. La declaración **const** no está permitida para constructores y destructores de objetos **const**. Un constructor debe tener permiso para modificar un objeto, de tal modo que pueda inicializarse de manera apropiada. Un destructor debe ser capaz de realizar sus funciones de limpieza final antes de destruir al objeto.



**Error común de programación 17.4**

*Intentar declarar un constructor o un destructor como **const**, es un error de sintaxis.*

En el código de la figura 17.1 creamos dos objetos de la clase **Hora**, un objeto no constante y un objeto constante. El programa intenta modificar el objeto **const mediodia** mediante las funciones miembro no constantes **estableceHora** (en la línea 102) e **imprimeEstandar** (en la línea 108). El programa también ilustra las otras tres combinaciones de las llamadas de funciones miembro a objetos; una función miembro no constante a un objeto no constante (línea 100), una función miembro **const** a un objeto no constante (línea 104) y una función miembro **const** a un objeto **const** (líneas 106 y 107). En la salida de ejemplo aparecen los mensajes que genera un popular compilador para la llamada de funciones miembro no constantes a objetos **const**.

```
1  // Figura 17.1: hora5.h
2  // Declaración de la clase Hora.
3  // Las funciones miembro están definidas en hora5.cpp
4  #ifndef HORA5_H
5  #define HORA5_H
6
7  class Hora {
```

**Figura 17.1** Uso de la clase **Hora** con objetos **const** y funciones miembro **const**; **hora5.h**. (Parte 1 de 4.)

```
public:
   Hora( int = 0, int = 0, int = 0 );    // constructor predeterminado

   // funciones establecer
   void estableceHora( int, int, int );  // establece hora
   void estableceHora( int );      // establece hora
   void estableceMinuto( int );     // establece minuto
   void estableceSegundo( int );    // establece segundo

   // funciones obtener (normalmente declaradas como const)
   int obtieneHora() const;       // devuelve hora
   int obtieneMinuto() const;     // devuelve minuto
   int obtieneSegundo() const;    // devuelve segundo

   // funciones imprime (normalmente declaradas como const)
   void imprimeMilitar() const;   // imprime hora militar
   void imprimeEstandar();        // imprime hora estándar
private:
   int hora;              // 0 - 23
   int minuto;            // 0 - 59
   int segundo;           // 0 - 59
}; // fin de la clase Hora

#endif
```

Figura 17.1 Uso de la clase **`Hora`** con objetos **`const`** y funciones miembro **`const`**; **`hora5.h`**. (Parte 1 de 5.)

```
// Figura 17.1: hora5.cpp
// Definiciones de las funciones miembro para la clase Hora.
#include <iostream>

using std::cout;

#include “hora5.h”

// Función constructor para inicializar datos privados.
// Los valores predeterminados son 0 (vea la definición de la clase).
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }

// Establece los valores de hora, minuto y segundo.
void Hora::estableceHora( int h, int m, int s )
{
   estableceHora( h );
   estableceMinuto( m );
   estableceSegundo( s );
} // fin de la función estableceHora

// Establece el valor de hora
void Hora::estableceHora( int h )
   { hora = ( h >= 0 && h < 24 ) ? h : 0; }

```

Figura 17.1 Uso de la clase **`Hora`** con objetos **`const`** y funciones miembro **`const`**; **`hora5.cpp`**. (Parte 2 de 5.)

```
// Establece el valor de minuto
void Hora::estableceMinuto( int m )
   { minuto = ( m >= 0 && m < 60 ) ? m : 0; }

// Establece el valor de segundo
void Hora::estableceSegundo( int s )
   { segundo = ( s >= 0 && s < 60 ) ? s : 0; }

// Obtiene el valor de hora
int Hora::obtieneHora() const { return hora; }

// Obtiene el valor de minuto
int Hora::obtieneMinuto() const { return minuto; }

// Obtiene el valor de segundo
int Hora::obtieneSegundo() const { return segundo; }

// Despliega la hora en formato militar: HH:MM
void Hora::imprimeMilitar() const
{
   cout << ( hora < 10 ? “0” : “” ) << hora << “:”
        << ( minuto < 10 ? “0” : “” ) << minuto;
} // fin de la función imprimeMilitar

// Despliega la hora en formato estándar: HH:MM:SS AM (o PM)
void Hora::imprimeEstandar()  // debe ser const
{
   cout << ( ( hora == 12 ) ? 12 : hora % 12 )   << “:”
        << ( minuto < 10 ? “0” : “” )  << minuto << “:”
        << ( segundo < 10 ? “0” : “” ) << segundo
        << ( hora < 12 ? “ AM” : “ PM” );
} // fin de la función imprimeEstandar
```

**Figura 17.1** Uso de la clase **Hora** con objetos **const** y funciones miembro **const**; **hora5.cpp**. (Parte 3 de 5.)

```
// Figura 17.1: fig17_01.cpp
// Intento de acceder a un objeto constante con
// funciones miembro no constantes.
#include “hora5.h”

int main()
{
   Hora levantarse( 6, 45, 0 );        // objeto no constante
   const Hora mediodia( 12, 0, 0 );    // objeto constante

                              // FUNCIÓN MIEMBRO OBJETO
 levantarse.estableceHora( 18 );   // no constante        no constante

   mediodia.estableceHora( 12 );   // no constante        constante

   levantarse.obtieneHora();       // constante           no constante

   mediodia.obtieneMinuto();       // constante           constante
```

**Figura 17.1** Uso de la clase **Hora** con objetos **const** y funciones miembro **const**; **fig17_01.cpp**. (Parte 4 de 5.)

```
107     mediodia.imprimeMilitar();     // constante          constante
108     mediodia.imprimeEstandar();    // no constante       constante
109     return 0;
110  } // fin de la función main
```

*Mensajes de error del compilador Visual C++ de Microsoft*

```
c:\ fig17_01.cpp(14) : error C2663: 'estableceHora' : 2 overloads have no legal
conversion for 'this' pointer
c:\ fig17_01.cpp(20) : error C2662: 'imprimeEstandar' : cannot convert 'this'
pointer from 'const class Hora' to 'class Hora &'
Conversion loses qualifiers
Error executing cl.exe.

fig17_01.obj - 2 error(s), 0 warning(s)
```

**Figura 17.1** Uso de la clase **`Hora`** con objetos **`const`** y funciones miembro **`const`**; **`fig17_01.cpp`**. (Parte 5 de 5.)



### Buena práctica de programación 17.1

*Declare como* **`const`** *todas las funciones miembro que no necesiten modificar el objeto actual, de modo que si lo requiere pueda utilizarlas en un objeto* **`const`**.

Observe que aun cuando un constructor debe ser una función miembro no constante, se le puede llamar para un objeto **`const`**. La definición del constructor **`Hora`** en las líneas 42 y 43

```
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }
```

muestra que el constructor **`Hora`** llama a la función miembro no constante **`estableceHora`** para realizar la inicialización de un objeto de **`Hora`**. Es válido invocar a una función miembro no constante desde la llamada al constructor de un objeto **`const`**. La constancia de un objeto se refuerza desde el momento en que el constructor completa la inicialización del objeto y hasta que se llama al destructor de dicho objeto.



### Observación de ingeniería de software 17.4

*Un objeto* **`const`** *no puede modificarse por asignación, por lo que es necesario inicializarlo. Cuando se declara un dato miembro de una clase como* **`const`**, *debe utilizarse un inicializador de miembro para proporcionar el constructor con el valor inicial del dato miembro del objeto de la clase.*

También observe que la línea 108 (línea 20 en el archivo fuente)

```
mediodia.imprimeEstandar();   // no constante       constante
```

genera un error de compilación, aun cuando la función miembro **`imprimeEstandar`** de la clase **`Hora`** no modifica los objetos en los cuales se invoca. El hecho de que una función no modifique un objeto no es suficiente para indicar un método **`const`**.

La figura 17.2 muestra el uso de un inicializador de miembro para inicializar el dato miembro **`const incremento`** de la clase **`Incremento`**. El constructor para **`Incremento`** se modifica de la siguiente manera:

```
Incremento::Incremento( int c, int i )
   : incremento( i )
{ cuenta = c; }
```

La notación **`:incremento( i )`** inicializa **`incremento`** en el valor **`i`**. Si se requieren varios inicializadores, simplemente inclúyalos en una lista separada por comas después de los dos puntos. Todos los datos miembro *pueden* inicializarse mediante el uso de la sintaxis de inicialización de miembros, pero los datos miembro **`const`** y las referencias *deben* inicializarse de esta manera. Más adelante, veremos que los objetos miembro deben inicializarse de esta manera. En el capítulo 19, cuando expliquemos la herencia, veremos que las porciones de la clase base de las clases derivadas también deben inicializarse de esta manera.

```
// Figura 17.2: fig17_02.cpp
// Uso de un inicializador de miembros para inicializar una
// constante de un tipo de dato predefinido.
#include <iostream>

using std::cout;
using std::endl;

class Incremento {
public:
   Incremento( int c = 0, int i = 1 );
   void sumaIncremento() { cuenta += incremento; }
   void imprime() const;

private:
   int cuenta;
   const int incremento;    // dato miembro const
}; // fin de la clase Incremento

// Constructor para la clase Incremento
Incremento::Incremento( int c, int i )
   : incremento( i )   // inicializador para el miembro const
{ cuenta = c; }

// Imprime los datos
void Incremento::imprime() const
{
   cout << “cuenta = “ << cuenta
        << “, incremento = “ << incremento << endl;
} // fin de la función imprime

int main()
{
   Incremento valor( 10, 5 );

   cout << “Antes del incremento: “;
   valor.imprime();

   for ( int j = 0; j < 3; j++ ) {
      valor.sumaIncremento();
      cout << “Despues del incremento “ << j + 1 << “: “;
      valor.imprime();
   } // fin de for

   return 0;
} // fin de la función main
```

```
Antes del incremento: cuenta = 10, incremento = 5
Despues del incremento 1: cuenta = 15, incremento = 5
Despues del incremento 2: cuenta = 20, incremento = 5
Despues del incremento 3: cuenta = 25, incremento = 5
```

**Figura 17.2** Uso de un inicializador de miembros para inicializar una constante de un tipo predefinido.

**Tip para prevenir errores 17.1**

*Siempre declare las funciones miembro como* **`const`**, *si no modifican el objeto. Esto puede ayudar a eliminar errores.*

La figura 17.3 muestra los errores de compilación generados por un popular compilador de C++ para un programa que intenta inicializar incremento con una instrucción de asignación en lugar de un inicializador de miembros.



**Error común de programación 17.5**

*No proporcionar un inicializador de miembros para un dato miembro* **`const`**, *es un error de sintaxis.*

```
// Figura 17.3: fig17_03.cpp
// Intento de inicializar una constante de un
// tipo de dato predefinido con una asignación.
#include <iostream>

using std::cout;
using std::endl;

class Incremento {
public:
   Incremento( int c = 0, int i = 1 );
   void sumaIncremento() { cuenta += incremento; }
   void imprime() const;
private:
   int cuenta;
   const int incremento;
}; // fin de la clase Incremento

// Constructor para la clase Incremento
Incremento::Incremento( int c, int i )
{              // El miembro constante ‘incremento’ no está inicializado
   cuenta = c;
   incremento = i;  // ERROR: No se puede modificar un objeto constante
} // fin del constructor Incremento

// Imprime los datos
void Incremento::imprime() const
{
   cout << “cuenta = “ << cuenta
         << “, incremento = “ << incremento << endl;
} // fin de la función imprime

int main()
{
 Incremento valor( 10, 5 );

   cout << “Antes del incremento: “;
   valor.imprime();

   for ( int j = 0; j < 3; j++ ) {
      valor.sumaIncremento();
      cout << “Despues del incremento “ << j << “: “;
      valor.imprime();
```

**Figura 17.3** Intento erróneo de inicializar por asignación una constante de un tipo predefinido. (Parte 1 de 2.)

```
44      } // fin de for
45
46      return 0;
47   } // fin de la función main
```

*Mensajes de error del compilador Visual C++ de Microsoft*

```
Compiling...
fig17_03.cpp
C:\ fig17_03.cpp(21) : error C2758: 'incremento' : must be initialized in
constructor base/member initializer list
C:\ fig17_03.cpp(16) : see declaration of 'incremento'
C:\ fig17_03.cpp(23) : error C2166: l-value specifies const object
Error executing cl.exe.

fig17_03.exe - 2 error(s), 0 warning(s)
```

**Figura 17.3** Intento erróneo de inicializar por asignación una constante de un tipo predefinido. (Parte 2 de 2.)



**Observación de ingeniería de software 17.5**

*Los miembros constantes de una clase (objetos y "variables" **const**) deben inicializarse con la sintaxis de inicialización de miembros; las asignaciones no están permitidas.*

Observe que la función **imprime** (línea 27) se declara como **const**. Es razonable, pero extraño, etiquetar esta función **const** debido a que probablemente nunca tendremos un objeto **const Incremento**.



**Observación de ingeniería de software 17.6**

*Es una buena práctica declarar como **const** a todas las funciones miembro de la clase que no modifican al objeto en el que operan. En algunas ocasiones, esto será una anomalía debido a que no tendrá la intención de crear objetos **const** de dicha clase. Declarar tales funciones miembro como **const** ofrece un gran beneficio. Si usted modifica inadvertidamente el objeto en esa función miembro, el compilador lanzará un mensaje de error de sintaxis.*



**Tip para prevenir errores 17.2**

*Los lenguajes como C++ son "blancos móviles" conforme evolucionan. Al lenguaje se adicionan más palabras reservadas. Evite utilizar palabras "cargadas", tales como "objeto", como identificadores. Aún cuando "objeto" no es una palabra reservada en C++, se podría convertir en una, de modo que la compilación con futuros compiladores podrían "romper" el código existente.*

## 17.3 Composición: Objetos como miembros de clases

Un objeto de la clase **AlarmaReloj** necesita saber cuándo se supone que debe sonar su alarma, ¿entonces por qué no incluir un objeto **Hora** como miembro del objeto **AlarmaReloj**? Tal capacidad se llama *composición*; una clase puede tener como miembros objetos de otra clase.



**Observación de ingeniería de software 17.7**

*La manera más común de reutilización de software es la composición, en la cual una clase tiene como miembros objetos de otras clases.*

Siempre que se crea un objeto, se invoca a su constructor, por lo que necesitamos especificar cómo se pasan los argumentos a los constructores del objeto miembro. Los objetos miembro se construyen en el orden en el que se declaran (no en el orden en el que aparecen en la lista del inicializador de miembros del constructor) y antes de que se construyan los objetos que los contienen (en ocasiones llamados objetos *host* [anfitriones]).

La figura 17.4 utiliza la clase **Empleado** y la clase **Fecha** para mostrar los objetos como miembros de otros objetos. La clase **Empleado** contiene los datos miembro privados **Nombre**, **Apellido**, **fechaNacimiento**, **fechaContratacion**. Los miembros **fechaNacimiento** y **fechaContratacion** son objetos **const** de la clase **Fecha**, la cual contiene los datos miembro privados **mes**, **dia** y **anio**. El progra-

ma crea la instancia del objeto **Empleado**, e inicializa y despliega sus datos miembro. Observe la sintaxis del encabezado de la función en la definición del constructor **Empleado**:

```
Empleado::Empleado( char *nomb, char *apell,
                    int mesnacim, int dianacim, int anionacim,
                    int mescontrat, int diacontrat, int aniocontrat )
    :fechaNacimiento( mesnacim, dianacim, anionacim ),
     fechaContratacion( mescontrat, diacontrat, aniocontrat )
```

El constructor toma ocho argumentos (**nomb**, **apell**, **mesnacim**, **dianacim**, **anionacim**, **mescontrat**, **diacontrat**, **aniocontrat**). Los dos puntos en el encabezado separan los inicializadores de miembros de la lista de parámetros. Los inicializadores de miembros especifican los argumentos de **Empleado** que se pasarán a los constructores de los objetos miembro **Fecha**. Los argumentos **mesnacim**, **dianacim** y **anionacim** se pasan al constructor del objeto **fechaNacimiento**, y los argumentos **mescontrat**, **diacontrat** y **aniocontrat** se pasan al constructor del objeto **fechaContratacion**. Los distintos inicializadores de miembros están separados por comas.

```
// Figura 17.4: fecha1.h
// Declaración de la clase Fecha.
// Las funciones miembro están definidas en fecha1.cpp
#ifndef FECHA1_H
#define FECHA1_H

class Fecha {
public:
   Fecha( int = 1, int = 1, int = 1900 ); // constructor predeterminado
   void imprime() const;  // imprime la fecha en formato mes/día/año
   ~Fecha();  // proporcionado para confirmar el orden de destrucción
private:
   int mes;   // 1-12
   int dia;   // 1-31 de acuerdo con el mes
   int anio;  // cualquier año

   // función de utilidad para verificar el día adecuado para el mes y el año
   int verificaDia( int );
}; // fin de la clase Fecha

#endif
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **fecha1.h**. (Parte 1 de 7.)

```
// Figura 17.4: fecha1.cpp
// Definiciones de las funciones miembro de la clase Fecha.
#include <iostream>

using std::cout;
using std::endl;

#include “fecha1.h”

// Constructor: Confirma el valor apropiado para el mes;
// llama a la función de utilidad verificaDia para confirmar el valor
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **fecha1.cpp**. (Parte 2 de 7.)

```
// apropiado para el día.
Fecha::Fecha( int mm, int dd, int aa )
{
   if ( mm > 0 && mm <= 12 )        // valida el mes
      mes = mm;
   else {
      mes = 1;
      cout << “Mes “ << mm << “ no valido. Establece en 1 al mes .\n”;
   } // fin de else

   anio = aa;                        // debe validar aa
   dia = verificaDia( dd );          // valida el día

   cout << “Constructor del objeto Fecha para la fecha “;
   imprime();          // interesante: una función imprime sin argumentos
   cout << endl;
} // fin del constructor Fecha

// Imprime el objeto Fecha en la forma mes/día/año
void Fecha::imprime() const
   { cout << mes << ‘/’ << dia << ‘/’ << anio; }

// Destructor: proporcionado para confirmar el orden de destrucción
Fecha::~Fecha()
{
   cout << “Destructor del objeto Fecha para la fecha “;
   imprime();
   cout << endl;
} // fin del destructor Fecha

// Función de utilidad para confirmar el valor apropiado de dia
// de acuerdo con el mes y el año.
// El 2000 es un año bisiesto?
int Fecha::verificaDia( int pruebaDia )
{
   static const int diasPorMes[ 13 ] =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   if ( pruebaDia > 0 && pruebaDia <= diasPorMes[ mes ] )
      return pruebaDia;

   if ( mes == 2 &&       // Febrero: Verifica si es año bisiesto
        pruebaDia == 29 &&
        ( anio % 400 == 0 ||
         ( anio % 4 == 0 && anio % 100 != 0 ) ) )
      return pruebaDia;

   cout << “Dia “ << pruebaDia << “ no valido. Establece en 1 al dia.\n”;

   return 1;  // deja al objeto en estado consistente, si hay un
                 mal valor
} // fin de la función verificaDia
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **`fecha1.cpp`**.
(Parte 3 de 7.)

```
// Figura 17.4: emplead1.h
// Declaración de la clase Empleado.
// Las funciones miembro están definidas en vacia1.cpp
#ifndef EMPLEAD1_H
#define EMPLEAD1_H

#include “fecha1.h”

class Empleado {
public:
   Empleado( char *, char *, int, int, int, int, int, int );
   void imprime() const;
   ~Empleado();  // proporcionado para confirmar el orden de destrucción
private:
   char nombre[ 25 ];
   char apellido[ 25 ];
   const Fecha fechaNacimiento;
   const Fecha fechaContratacion;
}; // fin de la clase Empleado

#endif
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **emplead1.h**. (Parte 4 de 7.)

```
// Figura 17.4: emplead1.cpp
// Definiciones de las funciones miembro de la clase Empleado.
#include <iostream>

using std::cout;
using std::endl;

#include <cstring>
#include “emplead1.h”
#include “fecha1.h”

Empleado::Empleado( char *nomb, char *apell,
                     int mesnacim, int dianacim, int anionacim,
                     int mescontrat, int diacontrat, int aniocontrat )
   : fechaNacimiento( mesnacim, dianacim, anionacim ),
     fechaContratacion( mescontrat, diacontrat, aniocontrat )
{
   // copia el nomb en nombre y verifica que coincide
   int longitud = strlen( nomb );
   longitud = ( longitud < 25 ? longitud : 24 );
   strncpy( nombre, nomb, longitud );
   nombre[ longitud ] = ‘\0’;

   // copia apell en apellido y se asegura de que coincide
   longitud = strlen( apell );
   longitud = ( longitud < 25 ? longitud : 24 );
   strncpy( apellido, apell, longitud );
   apellido[ longitud ] = ‘\0’;

   cout << “Constructor del objeto Empleado: “
        << nombre << ‘ ‘ << apellido << endl;
} // fin del constructor Empleado
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **emplead1.cpp**. (Parte 5 de 7.)

```

void Empleado::imprime() const
{
   cout << apellido << “, “ << nombre << “\nContratado: “;
   fechaContratacion.imprime();
   cout << “  Fecha de nacimiento: “;
   fechaNacimiento.imprime();
   cout << endl;
} // fin de la función imprime

// Destructor: proporcionado para confirmar el orden de destrucción
Empleado::~Empleado()
{
   cout << “Destructor del objeto Empleado: “
        << apellido << “, “ << nombre << endl;
} // fin del destructor Empleado
```

**Figura 17.4** Uso de los inicializadores de objetos miembro; **empleadl.cpp**. (Parte 6 de 7.)

```
// Figura 17.4: fig17_04.cpp
// Demostración de la composición: un objeto con objetos miembro.
#include <iostream>

using std::cout;
using std::endl;

#include “emplead1.h”

int main()
{
   Empleado e( “Roberto”, “Jimenez”, 7, 24, 1949, 3, 12, 1988 );

   cout << ‘\n’;
   e.imprime();

   cout << “\nPrueba el constructor Fecha con valores no validos:\n”;
   Fecha f( 14, 35, 1994 );  // valores inválidos de Fecha
   cout << endl;
   return 0;
} // fin de la función main
```

```
Constructor del objeto Fecha para la fecha 7/24/1949
Constructor del objeto Fecha para la fecha 3/12/1988
Constructor del objeto Empleado: Roberto Jimenez

Jimenez, Roberto
Contratado: 3/12/1988  Fecha de nacimiento: 7/24/1949

Prueba el constructor Fecha con valores no validos:
Mes 14 no valido. Establece en 1 al mes.
Dia 35 no valido. Establece en 1 al dia.
Constructor del objeto Fecha para la fecha 1/1/1994

Destructor del objeto Fecha para la fecha 1/1/1994
Destructor del objeto Empleado: Jimenez, Roberto
Destructor del objeto Fecha para la fecha 3/12/1988
Destructor del objeto Fecha para la fecha 7/24/1949
```

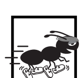
**Figura 17.4** Uso de los inicializadores de objetos miembro; **fig17_04.cpp**. (Parte 7 de 7.)

Recuerde que los miembros y las referencias **const** también se inicializan en la lista de inicialización de miembros (en el capítulo 19, veremos que las porciones de las clases base de clases derivadas también se inicializan de esta manera). Tanto la clase **Hora** como la clase **Empleado** incluyen una función destructora que imprime un mensaje cuando se destruye un objeto de **Fecha** o un objeto de **Empleado**, respectivamente. Esto nos permite confirmar en la salida del programa que los objetos se construyeron de adentro hacia afuera y se destruyeron en el orden inverso de afuera hacia adentro (es decir, los objetos miembro **Fecha** se destruyen después del objeto **Empleado** que los contiene).

Un objeto miembro no necesita inicializarse de manera explícita a través del inicializador de miembros. Si no se proporciona un miembro, implícitamente se llamará al constructor predeterminado del objeto miembro. Los valores, si existe alguno, establecidos por el constructor predeterminado pueden ser ignorados por las funciones *establecer*. Sin embargo, para una inicialización compleja, este método puede requerir bastante trabajo y tiempo adicional.

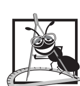
**Error común de programación 17.6**

*No proporcionar un constructor predeterminado para la clase de un objeto miembro, cuando no se proporciona un inicializador de miembros para dicho objeto, es un error de sintaxis.*



**Tip de rendimiento 17.2**

*Inicialice explícitamente los objetos miembro, a través de inicializadores de miembros. Esto elimina la sobrecarga de "inicializaciones duplicadas" de objetos miembro (una cuando se llama al constructor predeterminado del objeto miembro y otra cuando se utilizan las funciones establecer para inicializar el objeto miembro).*



**Observación de ingeniería de software 17.8**

*Si una clase tiene como miembro un objeto de otra clase, hacer que dicho objeto miembro sea público, no viola el encapsulamiento ni el ocultamiento de información de los miembros privados del objeto miembro.*

Observe la función miembro **imprime** de **Fecha** de la línea 52. En C++, muchas funciones miembro de clases no requieren argumentos. Esto se debe a que cada función miembro contiene un manipulador implícito (en la forma de un apuntador) al objeto en el cual opera. En la sección 17.5, explicaremos el apuntador implícito (llamado ***this***).

En esta primera versión de la clase **Empleado** (para facilidad de los programadores), utilizamos dos arreglos de 25 caracteres para representar el nombre y el apellido del **Empleado**. Estos arreglos pueden desperdiciar memoria en aquellos nombres más cortos que 24 caracteres (recuerde, un carácter en cada arreglo es para el carácter de terminación nulo **'\0'** de la cadena). Además, los nombres mayores a 24 caracteres deben truncarse para que quepan dentro de estos arreglos de caracteres. Más adelante presentaremos otra versión de la clase **Empleado** que crea de manera dinámica el monto de espacio exacto para almacenar el nombre y el apellido.

## 17.4 Funciones y clases friend (amigas)

Una *función* **friend** (amiga) de una clase se define fuera del alcance de la clase, pero tiene derechos de acceso a los miembros privados (y protegidos, como veremos en el capítulo 19) de la clase. Una función o una clase completa puede declararse como función **friend** de otra clase.

Al utilizar funciones amigas podemos aumentar el rendimiento. Aquí mostramos un ejemplo mecánico de cómo trabaja una función amiga. Más adelante, utilizaremos las funciones amigas para sobrecargar operadores y utilizarlos con objetos de una clase para crear clases iteradoras. Los objetos de una clase iteradora se utilizan para seleccionar sucesivamente elementos o realizar una operación con elementos de un objeto de una clase contenedora (vea la sección 17.9). Los objetos de una clase contenedora son capaces de almacenar elementos. Con frecuencia, el uso de amigas es apropiado cuando las funciones miembro no pueden utilizarse para ciertas operaciones, como veremos en el capítulo 18.

Para declarar una función como amiga de una clase, anteceda la palabra reservada **friend** al prototipo de la función en la definición de la clase. Para declarar a la **ClaseDos** como amiga de la **ClaseUno**, coloque una declaración de la forma

```
friend class ClaseDos;
```

en la definición de la **ClaseUno**.



### Observación de ingeniería de software 17.9

*Aunque los prototipos para las funciones amigas aparecen en la definición de la clase, las amigas no son funciones miembro.*



### Observación de ingeniería de software 17.10

*Los conceptos* **`private`**, **`protected`** *y* **`public`** *de acceso a miembros no son relevantes para las declaraciones de amistad, de modo que este tipo de declaraciones pueden colocarse en cualquier parte de la definición de la clase.*



### Buena práctica de programación 17.2

*Inmediatamente después del encabezado de la clase coloque las declaraciones de amistad, y no las anteceda con algún especificador de acceso a miembros.*

La amistad se gana, no se toma, es decir, para que la clase **B** sea una amiga de la clase **A**, la clase **A** debe declarar de manera explícita que la clase **B** es su amiga. Además, la amistad no es ni simétrica ni transitiva, si la clase **A** es amiga de la clase **B**, y la clase **B** es amiga de la clase **C**, usted no puede inferir que la clase **B** es amiga de la clase **A** (de nuevo, la amistad no es simétrica), que la clase **C** es amiga de la clase **B** o que la clase **A** es amiga de la clase **C** (de nuevo, la amistad no es transitiva).



### Observación de ingeniería de software 17.11

*Algunas personas en la comunidad de la programación orientada a objetos sienten que la "amistad" corrompe el ocultamiento de información y debilita el valor del método de diseño orientado a objetos.*

La figura 17.5 muestra la declaración y el uso de la función amiga **`estableceX`** para establecer el dato miembro privado **`x`** de la clase **`cuenta`**. Observe que la declaración **`friend`** aparece primero (por convención) en la declaración de la clase, incluso antes de la declaración de las funciones miembro públicas. El programa de la figura 17.6 muestra los mensajes que produce el compilador cuando se llama a la función no amiga **`noPuedeEstablecerX`** para modificar el dato miembro **`x`**. Las figuras 17.5 y 17.6 tienen la intención de introducir la "mecánica" del uso de las funciones amigas; los ejemplos prácticos del uso de las funciones amigas aparecerán en los siguientes capítulos.

```
// Figura 17.5: fig17_05.cpp
// Las funciones amigas de una clase pueden acceder a miembros privados de la clase.
#include <iostream>

using std::cout;
using std::endl;

// Clase modificada Cuenta
class Cuenta {
   friend void estableceX( Cuenta &, int ); // declaración de la amiga
public:
   Cuenta() { x = 0; }                      // constructor
   void imprime() const { cout << x << endl; }  // salida
private:
   int x;  // dato miembro
}; // fin de la clase Cuenta

// Es posible mofificar datos privados de la clase Cuenta
// debido a que estableceX está declarada como una función amiga de Cuenta
void estableceX( Cuenta &c, int val )
{
   c.x = val;  // legal: estableceX es una amiga de Cuenta
```

**Figura 17.5** Las amigas pueden acceder a los datos privados de una clase. (Parte 1 de 2.)

```
} // fin de la función estableceX

int main()
{
   Cuenta contador;

   cout << "contador.x despues de crear la instancia: ";
   contador.imprime();
   cout << "contador.x despues de llamar a la funcion amiga estableceX: ";
   estableceX( contador, 8 );  // establece x con una amiga
   contador.imprime();
   return 0;
} // fin de la función main
```

```
contador.x despues de crear la instancia: 0
contador.x despues de llamar a la funcion amiga estableceX: 8
```

**Figura 17.5** Las amigas pueden acceder a los datos privados de una clase. (Parte 2 de 2.)

```
// Figura 17.6: fig17_06.cpp
// Las funciones no amigas/no miembros no pueden acceder
// a los datos privados de una clase.
#include <iostream>

using std::cout;
using std::endl;

// Clase Cuenta modificada
class Cuenta {
public:
   Cuenta() { x = 0; }                     // constructor
   void imprime() const { cout << x << endl; }  // salida
private:
   int x;  // dato miembro
}; // fin de la clase Cuenta

// La función intenta modificar datos privados de Cuenta,
// pero no puede debido a que no es una amiga de Cuenta.
void noPuedeEstablecerX( Cuenta &c, int val )
{
   c.x = val;  // ERROR: 'Cuenta::x' nos es accesible
} // fin de la función noPuedeEstablecerX

int main()
{
   Cuenta contador;

   noPuedeEstablecerX( contador, 3 ); // noPuedeEstablecerX no es una amiga
   return 0;
} // fin de la función main
```

**Figura 17.6** Las funciones no amigas/ no miembro no pueden acceder a los datos privados de una clase. (Parte 1 de 2.)

*Mensajes de error del compilador Visual C++ de Microsoft*

```
C:\ fig17_06.cpp(22) : error C2248: 'x' : cannot access private member
declared in class 'Cuenta'
        C:\ fig17_06.cpp(15) : see declaration of 'x'
Error executing cl.exe.
```

**Figura 17.6** Las funciones no **amigas**/ no miembro no pueden acceder a los datos privados de una clase. (Parte 2 de 2.)

Observe que la función **estableceX** (línea 20) es una función independiente al estilo C; no es una función miembro de la clase **Cuenta**. Por esta razón, cuando se invoca a **estableceX** desde el objeto contador, utilizamos la instrucción de la línea 32

```
estableceX( contador, 8 );   // establece x mediante una amiga
```

la cual toma a **contador** como un argumento, en lugar de utilizar un manipulador (tal como el nombre del objeto) para llamar a la función, como en

```
contador.estableceX( 8 );
```

Como lo mencionamos, la figura 17.5 es un ejemplo mecánico de la construcción de una **friend**. Por lo general, es apropiado definir a la función **estableceX** como una función miembro de la clase **Cuenta**.

**Observación de ingeniería de software 17.12**

*C++ es un lenguaje híbrido, de modo que es común tener una mezcla de dos tipos de llamadas a funciones dentro de un programa y con frecuencia uno después del otro; llamadas estilo C que pasan datos primitivos u objetos a funciones, y llamadas de C++ que pasan funciones (o mensajes) a objetos.*

Es posible especificar funciones sobrecargadas como amigas de una clase. Cada función sobrecargada que pretendamos sea una amiga, debe declararse de manera explícita en la definición de la clase como una amiga de dicha clase.

## 17.5 Uso del apuntador `this`

Todos los objetos tienen acceso a su propia dirección mediante un apuntador llamado **this**. El apuntador **this** de un objeto no es parte del objeto mismo, es decir, el apuntador **this** no se refleja en el resultado de una operación **sizeof** sobre el objeto. En vez de esto, el apuntador **this** pasa al objeto (lo hace el compilador) como el primer argumento implícito de cada llamada de una función miembro no estática al objeto (en la sección 17.7, explicaremos los miembros estáticos).

El apuntador **this** se utiliza implícitamente para hacer referencia tanto a los datos miembro como a las funciones miembro de un objeto; también puede utilizarse de manera explícita. El tipo del apuntador **this** depende del tipo de objeto y de si la función miembro en la que se utiliza se declara como **const**. En una función miembro no constante de la clase **Empleado**, el apuntador **this** es de tipo **Empleado * const** (un apuntador constante a un objeto de **Empleado**). En una función miembro constante de la clase **Empleado**, el apuntador **this** tiene un tipo de dato **const Empleado * const** (un apuntador constante hacia un objeto **Empleado** que es constante).

Por ahora, mostramos un ejemplo sencillo del uso explícito del apuntador **this**; más adelante en este capítulo y en el capítulo 18, mostraremos algunos ejemplos sutiles pero sustanciales del uso de **this**. Toda función miembro no estática tiene acceso al apuntador **this** que apunta al objeto para el cual se invocó a la función miembro.

**Tip de rendimiento 17.3**



*Por razones de economía de almacenamiento, sólo existe una copia de cada función miembro por clase, y esta función miembro es invocada por cada objeto de la clase. Por otra parte, cada objeto tiene su propia copia de los datos miembro de la clase.*

La figura 17.7 muestra el uso explícito del apuntador **this** para permitir a una función miembro de la clase **Prueba** imprimir el dato privado **x** de un objeto **Prueba**.

Con efectos ilustrativos, la función miembro **imprime** de la figura 17.7 primero imprime directamente a **x**. Luego, **imprime** utiliza dos notaciones diferentes para acceder a **x** a través del apuntador **this**; con el operador flecha (**->**) de acceso a miembros y con el operador punto (**.**) del apuntador **this** desreferenciado.

Observe los paréntesis alrededor de ***this**, cuando se utiliza con el operador de selección de miembros (**.**). Los paréntesis son necesarios debido a que el operador punto tiene una precedencia más alta que el operador *****. Sin los paréntesis, la expresión

```
*this.x
```

se evaluaría como si tuviera los paréntesis de la siguiente manera:

```
*( this.x )
```

lo cual es un error de sintaxis debido a que el operador punto no puede utilizarse con un apuntador.



### Error común de programación 17.7

*Intentar utilizar el operador de selección de miembros (**.**) con un puntador hacia un objeto, es un error de sintaxis; el operador punto de selección de miembros sólo puede utilizarse con un objeto o con una referencia a un objeto.*

```
// Figura 17.7: fig17_07.cpp
// Uso del apuntador this para hacer referencia a objetos miembro.
#include <iostream>

using std::cout;
using std::endl;

class Prueba {
public:
   Prueba( int = 0 );              // constructor predeterminado
   void imprime() const;
private:
   int x;
}; // fin de la clase Prueba

Prueba::Prueba( int a ) { x = a; }  // constructor

void Prueba::imprime() const   // Los ( ) alrededor de *this son necesarios
{
   cout << “        x = “ << x
        << “\n  this->x = “ << this->x
        << “\n(*this).x = “ << ( *this ).x << endl;
} // fin de la función imprime

int main()
{
   Prueba objetoPrueba( 12 );

   objetoPrueba.imprime();

   return 0;
} // fin de la función main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

**Figura 17.7** Uso del apuntador **this**.

Un uso interesante del apuntador **this** es para prevenir que un objeto se asigne a sí mismo. Como veremos en el capítulo 18, la autoasignación puede provocar errores serios cuando los objetos contienen apuntadores hacia áreas de memoria asignadas en forma dinámica.

Otro uso del apuntador **this** es el de permitir las llamadas de funciones en cascada. La figura 17.8 muestra el retorno de una referencia hacia un objeto de **Hora** para permitir que las llamadas a funciones miembro de la clase **Hora** se hagan en cascada. Las funciones miembro **estableceHora**, **establecehora**, **estableceMinuto** y **estableceSegundo** devuelven un ***this** con un tipo de retorno **Hora&**.

```
// Figura 17.8: hora6.h
// Llamadas a funciones miembro en cascada.

// Declaración de la clase Hora.
// Las funciones miembro están definidas en hora6.cpp
#ifndef HORA6_H
#define HORA6_H

class Hora {
public:
   Hora( int = 0, int = 0, int = 0 );  // constructor predeterminado

   // funciones establecer
   Hora &estableceHora( int, int, int ); // establece hora, minuto, segundo
   Hora &establecehora( int );     // establece hora
   Hora &estableceMinuto( int );   // establece minuto
   Hora &estableceSegundo( int );  // establece segundo

   // funciones obtener (normalmente declaradas como const)
   int obtienehora() const;        // devuelve hora
   int obtieneMinuto() const;      // devuelve minuto
   int obtieneSegundo() const;     // devuelve segundo

   // funciones imprime (normalmente declaradas como const)
   void imprimeMilitar() const;    // imprime la hora militar
   void imprimeEstandar() const;   // imprime la hora estándar
private:
   int hora;               // 0 - 23
   int minuto;             // 0 - 59
   int segundo;            // 0 - 59
}; // fin de la clase Hora

#endif
```

**Figura 17.8** Llamadas en cascada a funciones miembro; **hora6.h**. (Parte 1 de 5.)

```
// Figura 17.8: hora6.cpp
// Definiciones de las funciones miembro para la clase Hora.
#include <iostream>

using std::cout;

#include “hora6.h”

// Función constructor para inicializar datos privados.
// Llama a la función miembro estableceHora, para establecer variables.
```

**Figura 17.8** Llamadas en cascada a funciones miembro; **hora6.cpp**. (Parte 2 de 5.)

```
// Los valores predeterminados son 0 (vea la definición de la clase).
Hora::Hora( int hr, int min, int seg )
   { estableceHora( hr, min, seg ); }

// Establece los valores de hora, minuto y segundo.
Hora &Hora::estableceHora( int h, int m, int s )
{
   establecehora( h );
   estableceMinuto( m );
   estableceSegundo( s );
   return *this;   // permite la cascada
} // fin de la función estableceHora

// Establece el valor de hora
Hora &Hora::establecehora( int h )
{
   hora = ( h >= 0 && h < 24 ) ? h : 0;

   return *this;   // permite la cascada
} // fin de la función establecehora

// Establece el valor de minuto
Hora &Hora::estableceMinuto( int m )
{
   minuto = ( m >= 0 && m < 60 ) ? m : 0;

   return *this;   // permite la cascada
} // fin de la función estableceMinuto

// Establece el valor de segundo
Hora &Hora::estableceSegundo( int s )
{
   segundo = ( s >= 0 && s < 60 ) ? s : 0;

   return *this;   // permite la cascada
} // fin de la función estableceSegundo

// Obtiene el valor de hora
int Hora::obtienehora() const { return hora; }

// Obtiene el valor de minuto
int Hora::obtieneMinuto() const { return minuto; }

// Obtiene el valor de segundo
int Hora::obtieneSegundo() const { return segundo; }

// Despliega la hora en formato militar: HH:MM
void Hora::imprimeMilitar() const
{
   cout << ( hora < 10 ? "0" : "" ) << hora << ":"
        << ( minuto < 10 ? "0" : "" ) << minuto;
} // fin de la función imprimeMilitar

// Despliega la hora en formato estándar: HH:MM:SS AM (o PM)
void Hora::imprimeEstandar() const
{
```

**Figura 17.8** Llamadas en cascada a funciones miembro; **hora6.cpp**. (Parte 3 de 5.)

```
    cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
         << ":" << ( minuto < 10 ? "0" : "" )  << minuto
         << ":" << ( segundo < 10 ? "0" : "" ) << segundo
         << ( hora < 12 ? " AM" : " PM" );
} // fin de la función imprimeEstandar
```

**Figura 17.8** Llamadas en cascada a funciones miembro; **`hora6.cpp`**. (Parte 4 de 5.)

```
// Figura 17.8: fig17_08.cpp
// Llamadas a funciones miembro en cascada
// con el apuntador this
#include <iostream>

using std::cout;
using std::endl;

#include "hora6.h"

int main()
{
   Hora h;

   h.establecehora( 18 ).estableceMinuto( 30 ).estableceSegundo( 22 );
   cout << "Hora militar: ";
   h.imprimeMilitar();
   cout << "\nHora estandar: ";
   h.imprimeEstandar();

   cout << "\n\nNueva hora estandar: ";
   h.estableceHora( 20, 20, 20 ).imprimeEstandar();
   cout << endl;

   return 0;
} // fin de la función main
```

```
Hora militar: 18:30
Hora estandar: 6:30:22 PM

Nueva hora estandar: 8:20:20 PM
```

**Figura 17.8** Llamadas en cascada a funciones miembro; **`fig17_08.cpp`**. (Parte 5 de 5.)

¿Por qué funciona la técnica de retorno de **`*this`**? El operador (**`.`**) asocia de izquierda a derecha, de modo que la expresión

```
h.establecehora( 18 ).estableceMinuto( 30 ). estableceSegundo( 22 );
```

primero evalúa a **`h.establecehora( 18 )`** y devuelve una referencia al objeto **`h`** como el valor de la llamada a esta función. El resto de la expresión se interpreta como

```
estableceMinuto( 30 ). estableceSegundo( 22 );
```

La llamada a **`estableceMinuto( 30 )`** se ejecuta y devuelve el equivalente de **`h`**. El resto de la expresión se interpreta como

```
h.estableceSegundo( 22 );
```

Observe que las llamadas

```
h.estableceHora( 20, 20, 20 ).imprimeEstandar();
```

también utiliza el método de cascada. Estas llamadas deben aparecer en ese orden en esta expresión, debido a que **imprimeEstandar** como se definió en la clase no devuelve una referencia a **h**. Colocar una llamada a **imprimeEstandar** en la instrucción anterior, antes de la llamada a **estableceHora**, es un error de sintaxis.

## 17.6 Asignación dinámica de memoria mediante los operadores new y delete

Los operadores **new** y **delete** constituyen un mejor método para realizar la asignación dinámica de memoria (para cualquier tipo predefinido o definido por el usuario), que las llamadas a las funciones **malloc** y **free** de C. Considere el siguiente código

```
nombreTipo *ptrNombreTipo;
```

En ANSI C, para crear de forma dinámica un objeto de tipo **nombreTipo**, usted escribiría

```
ptrNombreTipo = malloc( sizeof( nombreTipo ) );
```

Esto requiere una llamada a la función **malloc** y el uso explícito del operador **sizeof**. En versiones de C previas al C de ANSI, usted también tendría que convertir el tipo del apuntador devuelto por **malloc** mediante el operador de conversión de tipo (**NombreTipo ***). La función **malloc** no proporciona método alguno para inicializar el bloque asignado de memoria. En C++, usted simplemente escribe

```
ptrNombreTipo = new NombreTipo;
```

El nuevo operador crea un objeto del tamaño apropiado, llama al constructor para el objeto, y devuelve un apuntador del tipo correcto. En las versiones previas al C++ estándar de ANSI/ISO, si **new** es incapaz de encontrar espacio, devuelve un apuntador **0**. [*Nota:* En el capítulo 23, le mostraremos cómo lidiar con las fallas de **new**, en el contexto del C++ estándar de ANSI/ISO. En especial, le mostraremos cómo es que **new** "arroja" una "excepción" y le mostraremos cómo "atrapar" esa excepción y cómo lidiar con ella.] En C++, para destruir el objeto y liberar el espacio para este objeto, usted debe utilizar el operador **delete** de la siguiente manera:

```
delete ptrNombreTipo;
```

C++ le permite proporcionar un *inicializador* para un objeto creado recientemente, como en

```
double *ptrCosa = new double( 3.14159);
```

el cual inicializa un nuevo objeto **double** en **3.14159**.

Es posible crear un arreglo entero de 10 elementos y asignárselo a **ptrArreglo** de la siguiente manera:

```
int *ptrArreglo = new int[ 10 ];
```

Este arreglo se borra con la instrucción

```
delete [] ptrArreglo;
```

Como veremos, el uso de **new** y **delete** en lugar de **malloc** y **free** también ofrece otros beneficios. En especial, **new** invoca al constructor y **delete** invoca al destructor de la clase.



**Error común de programación 17.8**

*Mezclar el estilo de asignación dinámica de memoria* ***new*** *y* ***delete*** *con el estilo de asignación dinámica de* ***malloc*** *y* ***free****, es un error de lógica. El espacio creado por* ***malloc*** *no puede liberarse mediante* ***delete****; los objetos creados con* ***new*** *no pueden eliminarse mediante* ***free****.*



**Error común de programación 17.9**

*Utilizar* ***delete*** *en lugar de* ***delete []*** *para arreglos puede provocar errores lógicos en tiempo de ejecución. Para evitar problemas, el espacio creado como un arreglo debe eliminarse con el operador* ***delete []****, y el espacio creado como un elemento individual debe eliminarse con el operador* ***delete****.*



**Buena práctica de programación 17.3**

*C++ incluye a C, así que los programas en C++ pueden contener el almacenamiento creado por* ***malloc*** *y eliminarlo con* ***free****, y los objetos creados por* ***new*** *pueden eliminarse con* ***delete****. Es mejor utilizar sólo* ***new*** *y* ***delete****.*

## 17.7 Clases miembro static (estáticas)

Cada objeto de una clase tiene su propia copia de todos los datos miembro de la clase. Una variable estática de una clase se utiliza por ésta y por muchas otras razones. Una variable estática de una clase representa información "intrínseca de la clase" (es decir, propia de la clase, no de un objeto específico de la clase). La declaración de un miembro estático comienza con la palabra reservada **static**.

Motivemos la necesidad de datos estáticos intrínsecos de la clase con un ejemplo de juego de video. Suponga que tenemos un juego de video con **Marcianos** y otras criaturas del espacio. Cada **Marciano** tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presentes al menos cinco **Marcianos**. Si están presentes menos de cinco, cada **Marciano** se acobarda. Así que cada **Marciano** necesita saber la **cuentaMarcianos**. Podríamos incluir **cuentaMarcianos** en cada instancia de la clase **Marciano** como un dato miembro. Si hacemos esto, entonces cada **Marciano** tendrá una copia por separado de los datos miembro, y cada vez que creemos un nuevo **Marciano** tendremos que actualizar el dato miembro **cuentaMarcianos** en cada objeto **Marciano**. Esto representa un desperdicio de espacio con las copias redundantes, y un desperdicio de tiempo para la actualización de las copias separadas. En vez de esto, declaramos la variable **cuentaMarcianos** como **static**. Esto hace de **cuentaMarcianos** un dato intrínseco de la clase. Cada **Marciano** puede ver a **cuentaMarcianos** como si fuera un dato miembro de **Marciano**, pero C++ sólo mantiene una copia estática de **cuentaMarcianos**. Esto ahorra espacio. Nosotros ahorramos tiempo al hacer que el constructor **Marciano** incremente el dato estático **cuentaMarcianos**. Existe una sola copia, de modo que tenemos que incrementar por separado las copias de **cuentaMarciano** para cada objeto **Marciano**.



**Tip de rendimiento 17.4**

*Utilice datos miembro estáticos para ahorrar espacio, cuando sea suficiente una sola copia de los datos.*

Aunque los datos miembro estáticos pueden parecerse a las variables globales, los datos miembro estáticos tiene alcance de clase, los miembros estáticos pueden ser públicos, privados o protegidos. Los datos miembro estáticos *deben* inicializarse *una vez* (y sólo una vez) con alcance de archivo. Se puede acceder a las clases miembro estáticas y públicas a través de cualquier objeto de dicha clase, o se puede acceder a ellas a través del nombre de la clase por medio del operador binario de resolución de alcance. Se debe acceder a los miembros privados, protegidos y estáticos de una clase a través de funciones miembro públicas de la clase o a través de amigas de la clase. Los miembros estáticos de la clase existen, incluso cuando no existen objetos de esa clase. Para acceder a un miembro estático de una clase estática y pública cuando no existen objetos de la clase, simplemente coloque como prefijo del dato miembro el nombre de la clase y el operador binario de resolución de alcance (**::**). Para acceder a una clase miembro privada o protegida y estática cuando no existen objetos de la clase, debe proporcionarse una función miembro pública y estática, y la función debe invocarse colocando como prefijo de su nombre el nombre de la clase y el operador binario de resolución de alcance.

La figura 17.9 muestra un dato miembro privado y estático, y una función miembro pública y estática. El dato miembro cuenta se inicializa en cero y con alcance de archivo mediante la instrucción

```
int Empleado::cuenta = 0;
```

El dato miembro **cuenta** mantiene la cuenta del número de objetos de la clase **Empleado** que se tienen que instanciar. Cuando existen los objetos de la clase **Empleado**, puede utilizarse una referencia al miembro cuenta a través de cualquier función miembro de un objeto **Empleado**; en este ejemplo, tanto el constructor como el destructor hacen referencia a **cuenta**.

```
// Figura 17.9: emplead1.h
// Una clase empleado
#ifndef EMPLEAD1_H
#define EMPLEAD1_H

```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **emplead1.h**. (Parte 1 de 6.)

```
class Empleado {
public:
   Empleado( const char*, const char* );  // constructor
   ~Empleado();                           // destructor
   const char *obtieneNombre() const;     // devuelve el nombre
   const char *obtieneApellido() const;   // devuelve el apellido

   // función miembro estática
   static int obtieneCuenta();  // devuelve el # objetos instanciados

private:
   char *Nombre;
   char *Apellido;

   // dato miembro estático
   static int cuenta;  // número de objetos instanciados
}; // fin de la clase Empleado

#endif
```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **emplead1.h**. (Parte 2 de 6.)

```
// Figura 17.9: emplead1.cpp
// Definiciones de las funciones miembro para la clase Empleado
#include <iostream>

using std::cout;
using std::endl;

#include <cstring>
#include <cassert>
#include “emplead1.h”

// Inicializa el dato miembro estático
int Empleado::cuenta = 0;

// Define la función miembro estática que
// devuelve el número de objetos empleado instanciados.
int Empleado::obtieneCuenta() { return cuenta; }

// El constructor asigna de manera dinámica espacio para el
// nombre y el apellido, y utiliza strcpy para copiar
// el nombre y el apellido en el objeto
Empleado::Empleado( const char *nom, const char *apell )
{
   Nombre = new char[ strlen( nom ) + 1 ];
   assert( Nombre != 0 );   // garantiza la memoria asignada
   strcpy( Nombre, nom );

   Apellido= new char[ strlen( apell ) + 1 ];
   assert( Apellido != 0 );    // garantiza la memoria asignada
   strcpy( Apellido, apell );
```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **emplead1.cpp**. (Parte 3 de 6.)

```

   ++cuenta;  // incrementa la cuenta estática de empleados
   cout << “Constructor de Empleado para “ << Nombre
        << ‘ ‘ << Apellido << “ llamado.” << endl;
} // Fin del constructor Empleado

// El destructor desasigna la memoria asignada dinámicamente
Empleado::~Empleado()
{
   cout << “~Empleado() llamado para “ << Nombre
        << ‘ ‘ << Apellido << endl;
   delete [] Nombre;   // recaptura memoria
   delete [] Apellido; // recaptura memoria
   --cuenta;  // disminuye la cuenta estática de empleados
} // fin del destructor Empleado

// Devuelve el nombre del empleado
const char *Empleado::obtieneNombre() const
{
   // La constante antes del tipo de retorno previene al cliente de modificar
   // datos privados. El cliente debe copiar la cadena devuelta antes
   // de que el destructor borre la memoria para evitar un apuntador
      indefinido.
   return Nombre;
} // fin de la función obtieneNombre

// Devuelve el apellido del empleado
const char *Empleado::obtieneApellido() const
{
   // La constante antes del tipo de retorno previene al cliente de modificar
   // datos privados. El cliente debe copiar la cadena devuelta antes
   // de que el destructor borre la memoria para evitar un apuntador
      indefinido.
   return Apellido;
} // fin de la función obtieneApellido
```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **`emplead1.cpp`**. (Parte 4 de 6.)

```
// Figura 17.9: fig17_09.cpp
// Controlador para probar la clase empleado
#include <iostream>

using std::cout;
using std::endl;

#include “emplead1.h”

int main()
{
   cout << “El numero de empleados antes de crear la instancia es “
        << Empleado::obtieneCuenta() << endl;   // utiliza la clase nombre

```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **`fig17_09.cpp`**. (Parte 5 de 6.)

```
102     Empleado *ptrE1 = new Empleado( “Susana”, “Baez” );
103     Empleado *ptrE2 = new Empleado( “Roberto”, “Jimenez” );
104
105     cout << “El numero de empleados despues de crear la instancia es “
106          << ptrE1->obtieneCuenta();
107
108     cout << “\n\nEmpleado 1: “
109          << ptrE1->obtieneNombre()
110          << “ “ << ptrE1->obtieneApellido()
111          << “\nEmpleado 2: “
112          << ptrE2->obtieneNombre()
113          << “ “ << ptrE2->obtieneApellido() << “\n\n”;
114
115     delete ptrE1;   // recaptura memoria
116     ptrE1 = 0;
117     delete ptrE2;   // recaptura memoria
118     ptrE2 = 0;
119
120     cout << “El numero de empleados despues de la eliminacion es “
121          << Empleado::obtieneCuenta() << endl;
122
123     return 0;
124 } // fin de la función main
```

```
El numero de empleados antes de crear la instancia es 0
Constructor de Empleado para Susana Baez llamado.
Constructor de Empleado para Roberto Jimenez llamado.
El numero de empleados despues de crear la instancia es 2

Empleado 1: Susana Baez
Empleado 2: Roberto Jimenez

~Empleado() llamado para Susana Baez
~Empleado() llamado para Roberto Jimenez
El numero de empleados despues de la eliminacion es 0
```

**Figura 17.9** Uso de un dato miembro estático para mantener la cuenta del número de objetos de una clase; **`fig17_09.cpp`**. (Parte 6 de 6.)

### Error común de programación 17.10

*Es un error de sintaxis incluir la palabra reservada **`static`** en la definición de una variable estática de clase con alcance de archivo.*

Cuando no existen objetos de la clase **`Empleado`**, también se puede hacer referencia al miembro **`cuenta`**, pero solamente a través de una llamada a la función miembro estática **`obtieneCuenta`** de la siguiente manera:

```
Empleado::obtieneCuenta()
```

En este ejemplo, la función **`obtieneCuenta`** se utiliza para determinar el número de objetos instanciados de **`Empleado`**. Observe que cuando no existen objetos instanciados en el programa, se lanza la llamada a la función **`Empleado::obtieneCuenta()`**. Sin embargo, cuando existen objetos instanciados, se puede llamar a la función **`obtieneCuenta`** a través de uno de los objetos, como lo muestran las instrucciones de las líneas 105 y 106

```
cout << “El numero de empleados despues de crear la instancia es “
     << ptrE1->obtieneCuenta();
```

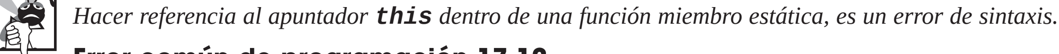
Observe que las llamadas **ptrE2->obtieneCuenta()** y **Empleado::obtieneCuenta()** producen el mismo resultado.



**Observación de ingeniería de software 17.13**

*Algunas empresas tienen en sus propios estándares de ingeniería de software que todas las llamadas a funciones miembro estáticas se hacen por medio del nombre de la clase y no con el manipulador de la clase.*

Una función miembro puede declararse como estática si no accede a los datos y a las funciones miembro no estáticos de la clase. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene un apuntador **this**, ya que los datos y las funciones miembro estáticas existen independientemente de cualquier objeto de la clase.



**Error común de programación 17.11**

*Hacer referencia al apuntador **this** dentro de una función miembro estática, es un error de sintaxis.*

**Error común de programación 17.12**

*Declarar una función miembro estática como **const**, es un error de sintaxis.*



**Observación de ingeniería de software 17.14**

*Los datos miembro estáticos y las funciones miembro estáticas existen y pueden utilizarse incluso si no se crean instancias de los objetos de esa clase.*

Las líneas 102 y 103 utilizan el operador **new** para asignar de manera dinámica dos objetos **Empleado**. Cuando se almacena cada objeto **Empleado**, se llama a su constructor. Cuando se utiliza **delete** en las líneas 115 y 117 para eliminar los dos objetos **Empleado**, se llama a sus destructores.



**Buena práctica de programación 17.4**

*Después de eliminar memoria asignada de manera dinámica, establezca en **0** al apuntador que hace referencia a dicha memoria. Esto desconecta al apuntador del espacio asignado previamente en el espacio libre.*

Observe el uso de **assert** en la función constructora **Empleado**. La “macro” **assert**, definida en el archivo de encabezado **cassert**, evalúa el valor de una condición. Si el valor de la expresión es **false**, entonces **assert** emite un mensaje de error y llama a la función **abort** (del archivo de encabezado de utilidades generales, **cstdlib**) para terminar la ejecución del programa. Ésta es una técnica de depuración útil para probar si una variable contiene el valor correcto. [*Nota:* La función **abort** termina de inmediato la ejecución del programa sin ejecutar destructor alguno.]

En este programa, **assert** determina si el operador **new** fue capaz de satisfacer el requerimiento de asignación dinámica de memoria. Por ejemplo, en la función constructora **Empleado**, la línea siguiente (a la cual se llama también *aserción*)

```
assert ( Nombre != 0 );
```

evalúa el apuntador **Nombre** para determinar si no es igual a **0**. Si la condición de la aserción anterior es **true**, el programa continúa sin interrupción. Si la condición en la aserción anterior es **false**, se imprime un mensaje de error que contiene el número de línea, la condición evaluada y el nombre del archivo en el cual aparece la aserción, y el programa termina. Entonces, el programador puede concentrarse en esta área de código para encontrar el error. En el capítulo 23, proporcionaremos un mejor método para lidiar con los errores en tiempo de ejecución.

Las aserciones no tienen que eliminarse del programa cuando termina la depuración. Cuando las aserciones ya no son necesarias para propósitos de depuración en un programa, la línea

```
#define NDEBUG
```

se inserta al principio del archivo de programa (por lo general esto también puede especificarse en la opciones del compilador). Esto provoca que el procesador ignore todas las aserciones, en lugar de que el programador tenga que borrar cada aserción manualmente.

Observe que la implementación de las funciones **obtieneNombre** y **obtieneApellido** devuelve al cliente de la clase apuntadores constantes a caracteres. En esta implementación, si el cliente desea retener una

copia del nombre o del apellido, el cliente es responsable de copiar la memoria asignada de manera dinámica en el objeto **Empleado**, después de obtener el apuntador constante al carácter desde el objeto. Observe que también es posible implementar **obtieneNombre** y **obtieneApellido** de modo que el cliente tenga que pasar un arreglo de caracteres y el tamaño del arreglo a cada función. Entonces, las funciones podrían copiar el nombre o el apellido dentro del arreglo de caracteres proporcionados por el cliente.

## 17.8 Abstracción de datos y ocultamiento de información

Por lo general, las clases ocultan sus detalles de implementación a sus clientes. A esto se le conoce como *ocultamiento de información*. Como un ejemplo del ocultamiento de información, consideremos una estructura de datos llamada *pila*.

Piense en una pila en términos de una pila de platos. Cuando se coloca un plato en la pila, siempre se coloca en la cima (a esto se le conoce como *empujar dentro de la pila (push)*), y cuando se quita un plato de la pila siempre se remueve de la cima (a esto se le conoce como *botar de la pila (pop)*). Las pilas se conocen como estructuras de *último en entrar, primero en salir (LIFO)*, el último elemento empujado (insertado) en la pila es el primer elemento botado (removido) de la pila.

El programador puede crear una clase pila y ocultar a sus clientes la implementación de dicha pila. Las pilas pueden implementarse muy fácil con arreglos (o con listas ligadas; vea el capítulo 12). Un cliente de la clase pila no necesita saber cómo se implementó dicha clase. El cliente sólo requiere que cuando los elementos de datos se coloquen en la pila, estos elementos sean llamados con el orden último en entrar, primero en salir. A la descripción de la funcionalidad de la clase, independientemente de su implementación, se le llama *abstracción de datos* y las clases en C++ definen los llamados *tipos de datos abstractos (ADTs)*. Aunque los usuarios pudieran conocer los detalles de la implementación de la clase, no deben escribir código que dependa de esos detalles. Esto significa que la implementación de una clase en particular (tal como una que implemente una pila y sus operaciones *push* y *pop*) puede alterarse o reemplazarse sin afectar al resto del sistema, mientras no se modifique la interfaz pública de la clase.

El trabajo de un lenguaje de alto nivel es crear una vista conveniente para el trabajo de los programadores. No existe un punto de vista estándar aceptado; ésta es una de las razones por las cuales existen tantos lenguajes de programación. La programación orientada a objetos en C++ presenta otro punto de vista.

La mayoría de los lenguajes de programación enfatizan las acciones. En estos lenguajes, los datos existen como soporte para las acciones que los programas necesitan llevar a cabo. De cualquier manera, los datos se ven como "menos interesantes" que las acciones. Los datos son "fríos". Sólo existen unos cuantos tipos de datos, y es difícil para los programadores crear sus propios tipos de datos.

Esta visión cambia con C++ y con la programación orientada a objetos. C++ eleva la importancia de los datos. La principal actividad en C++ es la creación de nuevos tipos de datos (es decir, clases), y expresar las interacciones entre objetos de dichos tipos.

Para moverse en esa dirección, la comunidad de los lenguajes de programación necesitaba formalizar algunos conceptos acerca de los datos. La formalización que nosotros consideramos es la de la idea de los tipos de datos abstractos (ADTs). En la actualidad, las ADTs reciben tanta atención como lo hizo la programación estructurada en las dos últimas décadas. Las ADTs no remplazan a la programación estructurada. En vez de eso, proporcionan una formalización adicional que puede mejorar el proceso de desarrollo de programas.

¿Qué es un tipo de dato abstracto? Considere un tipo predefinido como **int**. En lo primero que pensamos es en un entero; pero **int** en una computadora no es precisamente lo que es un entero en matemáticas. En particular, el **int** de la computadora es, por lo general, bastante limitado en tamaño. Por ejemplo, **int** en una máquina de 32 bits puede limitarse a un rango entre −2 miles de millones y + 2 miles de millones. Si el resultado de un cálculo cae fuera de este rango, ocurre un error de "desbordamiento" y la computadora responde de alguna manera dependiente del sistema, la cual incluye la posibilidad de producir "calladamente" un resultado incorrecto. Los enteros matemáticos no tienen este problema. Entonces, el concepto de un **int** en la computadora es en realidad sólo una aproximación del concepto de un entero en la realidad. Lo mismo es verdad con los **double**.

Incluso **char** es una aproximación; los valores **char** normalmente son patrones de ocho bits de unos y ceros; estos patrones no se parecen en nada a los caracteres que representan, como la **Z** mayúscula, la **z** minúscula,

un signo de moneda (**$**), un dígito (**5**), etcétera. En la mayoría de las computadoras, los valores de tipo **char** son bastante limitados si se les compara con el rango de caracteres reales. El conjunto de caracteres ASCII de siete bits proporciona 128 diferentes valores de caracteres. Esto es completamente inadecuado para representar lenguajes como el japonés y el chino que requieren miles de caracteres.

El punto es que incluso los tipos integrados provistos en los lenguajes de programación como C++ son en realidad sólo aproximaciones o modelos de conceptos y comportamientos reales. Hasta este punto hemos dado por hecho al tipo **int**, pero ahora tiene una nueva perspectiva a considerar. Los tipos como **int**, **double**, **char** y otros, son ejemplos de tipos de datos abstractos; en esencia, son formas de representar ideas reales hasta un cierto punto satisfactorio de precisión en un sistema de cómputo.

Un tipo de dato abstracto en realidad contempla dos ideas, a saber, la *representación de datos* y las *operaciones* que están permitidas con esos datos. Por ejemplo, en C++, la idea de **int** define las operaciones de suma, resta, multiplicación, división y módulo, pero la división entre cero no está definida; y estas operaciones permitidas se realizan de manera sensible a los parámetros de la máquina, como el tamaño de palabras fijas del sistema de cómputo subyacente. Otro ejemplo es la idea de los enteros negativos, cuyas operaciones y representación de datos son claras, pero la operación de obtener la raíz cuadrada de un entero negativo no está definida. En C++, el programador utiliza clases para implementar precisamente tipos de datos abstractos y sus servicios.

### 17.8.1 Ejemplo: Un tipo de dato abstracto Arreglo

En el capítulo 6 explicamos los arreglos. Un arreglo no es otra cosa que un apuntador y cierto espacio. Esta capacidad primitiva es aceptable para realizar operaciones con arreglos, si el programador es cuidadoso. Existen muchas operaciones que sería bueno realizar con arreglos, pero que no están integradas en C++. Con clases de C++, el programador puede desarrollar un tipo de dato abstracto Arreglo, el cual es preferible a los arreglos "puros". La clase arreglo puede proporcionar muchas nuevas capacidades útiles como

- Verificación del rango de subíndices.
- Un rango arbitrario de subíndices, en lugar de tener que iniciar con 0.
- Asignación de arreglos.
- Comparación de arreglos.
- Entrada/salida de arreglos.
- Arreglos que saben sus tamaños.
- Arreglos que se expanden dinámicamente para acomodar más elementos.

En el capítulo 18 creamos nuestra propia clase arreglo. C++ tiene un pequeño conjunto de tipos integrados. Las clases amplían al lenguaje de programación base.



**Observación de ingeniería de software 17.15**

*El programador puede crear nuevos tipos a través del mecanismo de la clase. Estos nuevos tipos pueden designarse para utilizarlos de manera tan conveniente como los tipos predefinidos. Por lo tanto, C++ es un lenguaje extensible. Aunque el lenguaje es fácil de ampliar con estos nuevos tipos, el lenguaje base en sí es modificable.*

Las nuevas clases creadas en ambientes de C++ pueden ser propiedad de un individuo, de pequeños grupos o de empresas. Las clases también pueden colocarse en bibliotecas de clases estándares, con la intención de distribuirlas. Esto no necesariamente promueve los estándares, aunque, de hecho, éstos están surgiendo. El valor completo de C++ puede apreciarse sólo cuando se utilizan bibliotecas de clases importantes u estandarizadas para desarrollar nuevas aplicaciones. ANSI (American National Standards Institute) e ISO (International Standards Organization) han desarrollado una versión estándar de C++ que incluye una biblioteca estándar de clases. El lector que aprende C++ y programación orientada a objetos estará listo para aprovechar los nuevos tipos de desarrollo rápido de software y orientado a componentes, que se hizo posible con las ricas y abundantes bibliotecas.

### 17.8.2 Ejemplo: Un tipo de dato abstracto Cadena

C++ es un lenguaje intencionalmente ralo que sólo proporciona a los programadores las capacidades puras necesarias para construir un amplio rango de sistemas (considérelo una herramienta para hacer herramientas). El

lenguaje está diseñado para minimizar las desventajas de rendimiento. C++ es adecuado tanto para programación de aplicaciones, como para programación de sistemas; esta última implica una extraordinaria demanda de rendimiento a los programas. Ciertamente, hubiera sido posible incluir un tipo de dato cadena entre los tipos predefinidos de C++. En su lugar, el leguaje se diseñó para que incluyera mecanismos para crear e implementar tipos de datos abstractos cadena a través de clases.

### 17.8.3 Ejemplo: Un tipo de dato abstracto Cola

De vez en cuando, todos nos formamos en una línea. A una línea de espera también se le llama *cola*. Hacemos cola en la caja registradora del supermercado, en la gasolinera, para abordar el autobús, para pagar el peaje en la autopista, y los estudiantes saben muy bien acerca de hacer cola para registrarse en los cursos que desean tomar. Los sistemas de cómputo utilizan muchas líneas de espera, por lo que necesitamos escribir programas que simulen lo que son y lo que hacen las colas.

Una cola es un buen ejemplo de un tipo de dato abstracto. Una cola ofrece un comportamiento bastante comprensible para sus clientes. Los clientes colocan elementos en una cola, uno a la vez (*enqueue)*, y los clientes toman (sacan) esos elementos de regreso, uno a la vez (*dequeue)*. Conceptualmente, una cola puede ser muy larga. Por supuesto, una cola real es finita. Los elementos se devuelven de una fila en orden *primero en entrar, primero en salir (PEPS)*; el primer elemento insertado en la cola es el primer elemento removido de ella.

La cola esconde una representación de datos interna que de alguna manera da seguimiento a los elementos que actualmente esperan en la línea, y ofrece un conjunto de operaciones a sus clientes, a saber, *enqueue* y *dequeue*. Los clientes no se preocupan por la implementación de la cola. Los clientes simplemente desean que la cola opere como "se publicó". Cuando un cliente forma un nuevo elemento en la cola, ésta debe aceptar dicho elemento y colocarlo en la fila con alguna estructura de datos con el orden de primero en entrar, primero en salir. Cuando el cliente quiere el siguiente elemento de la cola, ésta debe eliminar el elemento de su representación interna y debe entregarlo al mundo exterior (es decir, al *cliente* de la cola) en orden PEPS, es decir, el elemento que estuvo en la cola el mayor tiempo, debe ser el siguiente elemento devuelto por la operación *dequeue*.

El ADT cola garantiza la integridad de su estructura de datos interna. Los clientes no manipulan la estructura de datos de manera directa. Sólo las funciones miembro tienen acceso a sus datos internos. Los clientes únicamente pueden ocasionar la ejecución de operaciones permitidas en la representación de datos; las operaciones no proporcionadas en la interfaz pública de la ADT se rechazan de alguna manera adecuada. Esto podría significar la emisión de un mensaje de error, terminar la ejecución o simplemente ignorar la petición de la operación.

## 17.9 Clases contenedoras e iteradores

Entre los tipos de clases más populares están las *clases contenedoras* (también llamadas *colección de clases*), es decir, las clases diseñadas para almacenar colecciones de objetos. Por lo general, las clases contenedoras proporcionan servicios tales como la inserción, eliminación, búsqueda, ordenamiento, prueba de un elemento para determinar si es un miembro de la colección y otras cosas por el estilo. Arreglos, pilas, colas, árboles y listas ligadas son ejemplos de clases contenedoras.

Es común asociar *objetos iteradores*, o simplemente *iteradores*, con clases contenedoras. Un iterador es un objeto que devuelve el siguiente elemento de la colección (o realiza alguna acción en el siguiente elemento de una colección). Una vez que se escribe el iterador para las clases, y obtener el siguiente elemento desde la clase se puede expresar de manera sencilla. Tal como un libro que se comparte con varias personas y puede tener varias marcas al mismo tiempo, una clase contenedora puede tener varios iteradores operando al mismo tiempo. Cada iterador mantiene su propia "posición" en la información.

## *RESUMEN*

- La palabra reservada **`const`** especifica que un objeto no es modificable.
- El compilador de C++ no permite llamadas a funciones miembro no **`const`** en objetos **`const`**.

- Intentar modificar un objeto de una clase mediante una función miembro **const** de dicha clase, es un error de sintaxis.
- Una función se especifica como **const** tanto en su declaración como en su definición.
- Una función miembro **const** puede sobrecargarse con una versión no **const**. La elección con respecto a la función miembro a utilizar la hace el compilador, basándose en si el objeto fue declarado o no como **const**.
- Un objeto **const** debe inicializarse; los inicializadores de miembros deben proporcionarse dentro del constructor de la clase, cuando dicha clase contiene datos miembro **const**.
- Las clases pueden estar compuestas por objetos de otras clases.
- Los objetos miembro se construyen en el orden en el que se listan en la definición de la clase, y antes de que se construyan los objetos de su clase contenedora.
- Si no se proporciona un inicializador de miembros para un objeto miembro, se invoca al constructor predeterminado del objeto miembro.
- Una función **friend** (amiga) de una clase es una función definida fuera de esa clase y que tiene derecho de acceso a todos los miembros de la clase.
- Las declaraciones de amistad pueden colocarse en cualquier parte de la definición de la clase.
- El apuntador **this** se utiliza de manera implícita para hacer referencia tanto a funciones miembro no estáticas, como a datos miembro no estáticos del objeto.
- Cada función miembro no estática tiene acceso a la dirección de su objeto por medio de la palabra reservada **this**.
- El apuntador **this** puede utilizarse de manera explícita.
- El operador **new** asigna espacio para un objeto, ejecuta el constructor del objeto y devuelve un apuntador del tipo adecuado. Para liberar el espacio de este objeto, utilice el operador **delete**.
- Un arreglo de objetos puede asignarse de manera dinámica con **new** de la siguiente manera:

```
int *ptr = new int[ 100 ];
```

  el cual almacena un arreglo de 100 enteros y asigna la ubicación inicial del arreglo a **ptr**. El arreglo de enteros anterior se elimina con la instrucción:

```
delete [] ptr;
```

- Un dato miembro estático representa información “intrínseca de la clase” (es decir, propia de la clase, no de un objeto). La declaración de un miembro estático comienza con la palabra reservada **static**.
- Los datos miembro estáticos tienen alcance de clase.
- Se puede acceder a los miembros estáticos de una clase a través de un objeto de dicha clase, o a través del nombre de la clase por medio del operador de resolución de alcance (si el miembro es público).
- Una función miembro puede declararse como estática si no accede a los miembros no estáticos de la clase. A diferencia de las funciones miembro no estáticas, una función miembro estática no contiene un apuntador **this**. Esto se debe a que los datos miembro estáticos y las funciones miembro estáticas existen independientemente de cualquier objeto de la clase.
- Por lo general, las clases ocultan sus detalles de implementación a sus clientes. A esto se le llama ocultamiento de información.
- A las pilas se les conoce como estructuras de datos de tipo último en entrar, primero en salir (UEPS), el último elemento *empujado* (insertado) en la pila es el primer elemento *eliminado* (removido) de la pila.
- A la descripción de la funcionalidad de una clase, independientemente de su implementación, se le llama abstracción de datos, y las clases en C++ definen los llamados tipos de datos abstractos (ADTs).
- C++ aumenta la importancia de los datos. La principal actividad en C++ es crear nuevos tipos de datos (es decir, clases), y expresar las interacciones entre objetos de dichos tipos de datos.
- Los tipos de datos abstractos son formas de representar dentro de un sistema de cómputo, conceptos del mundo real con un nivel satisfactorio de precisión.
- En realidad, un tipo de dato abstracto representa dos conceptos, a saber, una representación de los datos y las operaciones permitidas con dichos datos.
- C++ es un lenguaje extensible. Aunque el lenguaje es fácil de extender con estos nuevos tipos, el lenguaje base por sí mismo no se puede modificar.
- Con toda la intención, C++ es un lenguaje ralo que sólo proporciona al programador las capacidades básicas necesarias para construir un amplio número de sistemas. El lenguaje está diseñado para minimizar las desventajas de rendimiento.
- Los elementos de una cola se devuelven con un orden primero en entrar, primero en salir (PEPS); el primer elemento insertado en la cola es el primer elemento removido de ella.

- Las clases contenedoras (también llamadas colección de clases) están diseñadas para almacenar colecciones de objetos. Por lo general las clases contenedoras proporcionan servicios tales como inserción, eliminación, búsqueda, ordenamiento, prueba de la membresía de un elemento a la clase, etcétera.
- Es común asociar objetos iteradores, o sencillamente iteradores, con clases contenedoras. Un iterador es un objeto que devuelve el siguiente elemento de una colección (o realiza alguna acción en el siguiente elemento de una colección).
- Cuando la definición de una clase sólo utiliza un apuntador a otra clase, no se tiene que incluir el archivo de encabezado para la otra clase (la cual, por lo general, revelaría los datos privados de la clase) por medio de **`#include`**. Usted puede sencillamente declarar la otra clase como un tipo de dato mediante una declaración de clase **`forward`**, antes de utilizar el tipo dentro del archivo.
- El archivo de implementación que contiene las funciones miembro para la clase **`proxy`** es el único archivo que incluye el encabezado para la clase cuyos datos privados deseamos ocultar.
- Al cliente se le proporciona el archivo de implementación como un objeto precompilado, junto con el archivo de encabezado que incluye los prototipos de las funciones de los servicios proporcionados por la clase **`proxy`**.

## TERMINOLOGÍA

alcance de clase
apuntador **`this`**
clase **`friend`** (amiga)
composición
constructor
constructor de un objeto miembro
constructor predeterminado
contenedor
dato miembro **`static`** (estático)
*dequeue* (sacar de la cola)
destructor
destructor predeterminado
*enqueue* (poner en la cola)
especificadores de acceso a miembros
función **`friend`** (amiga)
función miembro **`const`**
función miembro **`static`** (estática)
inicializador de miembro
iterador
lenguaje extensible
llamadas en cascada a funciones miembro
objeto anfitrión (host)
objeto **`const`**
objeto miembro
objetos dinámicos
operaciones en un ADT
operador binario de resolución de alcance (**`::`**)
operador de selección de miembros (**`.`**)
operador **`delete`**
operador **`delete []`**
operador flecha de selección de miembros (**`->`**)
operador **`new`**
operador **`new []`**
*pop* (operación de pilas)
primero en entrar, primero en salir (PEPS)
principio del menor privilegio
programación basada en objetos
*push* (operación de pilas)
representaciones de datos
tipo de dato abstracto (ADT)
tipo de dato abstracto cola
tipo de dato abstracto pila
último en entrar, primero en salir (UEPS)

## ERRORES COMUNES DE PROGRAMACIÓN

**17.1** Definir como **`const`** una función miembro que modifica un dato miembro de un objeto, es un error de sintaxis.

**17.2** Definir como **`const`** una función miembro que llama a una función miembro no **`const`** de la clase en la misma instancia de la clase, es un error de sintaxis.

**17.3** Invocar a una función miembro no **`const`** en un objeto **`const`**, es un error de sintaxis.

**17.4** Intentar declarar un constructor o un destructor como **`const`**, es un error de sintaxis.

**17.5** No proporcionar un inicializador de miembros para un dato miembro **`const`**, es un error de sintaxis.

**17.6** No proporcionar un constructor predeterminado para la clase de un objeto miembro, cuando no se proporciona un inicializador de miembros para dicho objeto, es un error de sintaxis.

**17.7** Intentar utilizar el operador de selección de miembros (**`.`**) con un puntador hacia un objeto, es un error de sintaxis; el operador punto de selección de miembros sólo puede utilizarse con un objeto o con una referencia a un objeto.

**17.8** Mezclar el estilo de asignación dinámica de memoria **`new`** y **`delete`** con el estilo de asignación dinámica de **`malloc`** y **`free`**, es un error de lógica. El espacio creado por **`malloc`** no puede liberarse mediante **`delete`**; los objetos creados con **`new`** no pueden eliminarse mediante **`free`**.

**17.9** Utilizar **`delete`** en lugar de **`delete []`** para arreglos puede provocar errores lógicos en tiempo de ejecución. Para evitar problemas, el espacio creado como un arreglo debe eliminarse con el operador **`delete []`**, y el espacio creado como un elemento individual debe eliminarse con el operador **`delete`**.

**17.10** Es un error de sintaxis incluir la palabra reservada **`static`** en la definición de una variable estática de clase con alcance de archivo.

**17.11** Hacer referencia al apuntador `this` dentro de una función miembro estática, es un error de sintaxis.

**17.12** Declarar una función miembro estática como `const`, es un error de sintaxis.

## BUENAS PRÁCTICAS DEPROGRAMACIÓN

**17.1** Declare como `const` todas las funciones miembro que no necesiten modificar el objeto actual, de modo que si lo requiere pueda utilizarlas en un objeto `const`.

**17.2** Inmediatamente después del encabezado de la clase coloque las declaraciones de amistad, y no las anteceda con algún especificador de acceso a miembros.

**17.3** C++ incluye a C, así que los programas en C++ pueden contener el almacenamiento creado por `malloc` y eliminarlo con `free`, y los objetos creados por `new` pueden eliminarse con `delete`. Es mejor utilizar sólo `new` y `delete`.

**17.4** Después de eliminar memoria asignada de manera dinámica, establezca en `0` al apuntador que hace referencia a dicha memoria. Esto desconecta al apuntador del espacio asignado previamente en el espacio libre.

## TIPS DE RENDIMIENTO

**17.1** Declarar variables y objetos `const` no sólo es una práctica de ingeniería de software efectiva, también puede mejorar el rendimiento debido a que los sofisticados compiladores actuales pueden realizar ciertas optimizaciones sobre constantes, que no es posible realizar sobre variables.

**17.2** Inicialice de manera explícita los objetos miembro, a través de inicializadores de miembros. Esto elimina la sobrecarga de “inicializaciones duplicadas” de objetos miembro (una cuando se llama al constructor predeterminado del objeto miembro y otra cuando se utilizan las funciones establecer para inicializar el objeto miembro).

**17.3** Por razones de economía de almacenamiento, sólo existe una copia de cada función miembro por clase, y esta función miembro es invocada por cada objeto de la clase. Por otra parte, cada objeto tiene su propia copia de los datos miembro de la clase.

**17.4** Utilice datos miembro estáticos para ahorrar espacio, cuando sea suficiente una sola copia de los datos.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**17.1** Declarar un objeto como `const` ayuda a reforzar el principio del menor privilegio. Los intentos para modificar al objeto son captados en tiempo de compilación, en lugar de provocar errores en tiempo de ejecución.

**17.2** Utilizar `const` es crucial para el diseño apropiado de clases y para la codificación y diseño de programas.

**17.3** Una función miembro `const` puede sobrecargarse con una versión no `const`. La elección respecto a cuál función miembro sobrecargada utilizar la hace el compilador, basándose en si el objeto es o no `const`.

**17.4** Un objeto `const` no puede modificarse por asignación, por lo que es necesario inicializarlo. Cuando se declara un dato miembro de una clase como `const`, debe utilizarse un inicializador de miembro para proporcionar el constructor con el valor inicial del dato miembro del objeto de la clase.

**17.5** Los miembros constantes de una clase (objetos y “variables” `const`) deben inicializarse con la sintaxis de inicialización de miembros; las asignaciones no están permitidas.

**17.6** Es una buena práctica declarar como `const` a todas las funciones miembro de la clase que no modifican al objeto en el que operan. En algunas ocasiones, esto será una anomalía debido a que no tendrá la intención de crear objetos `const` de dicha clase. Declarar tales funciones miembro como `const` ofrece un gran beneficio. Si usted modifica inadvertidamente el objeto en esa función miembro, el compilador lanzará un mensaje de error de sintaxis.

**17.7** La manera más común de reutilización de software es la composición, en la cual una clase tiene como miembros objetos de otras clases.

**17.8** Si una clase tiene como miembro un objeto de otra clase, hacer que dicho objeto miembro sea público, no viola el encapsulamiento ni el ocultamiento de información de los miembros privados del objeto miembro.

**17.9** Aunque los prototipos para las funciones amigas aparecen en la definición de la clase, las amigas no son funciones miembro.

**17.10** Los conceptos `private`, `protected` y `public` de acceso a miembros no son relevantes para las declaraciones de amistad, de modo que este tipo de declaraciones pueden colocarse en cualquier parte de la definición de la clase.

**17.11** Algunas personas en la comunidad de la programación orientada a objetos sienten que la “amistad” corrompe el ocultamiento de información y debilita el valor del método de diseño orientado a objetos.

**17.12** C++ es un lenguaje híbrido, de modo que es común tener una mezcla de dos tipos de llamadas a funciones dentro de un programa y con frecuencia uno después del otro; llamadas estilo C que pasan datos primitivos u objetos a funciones, y llamadas de C++ que pasan funciones (o mensajes) a objetos.

**17.13** Algunas empresas tienen en sus propios estándares de ingeniería de software con respecto a que todas las llamadas a funciones miembro estáticas se hacen por medio del nombre de la clase y no con el manipulador de la clase.

**17.14** Los datos miembro estáticos y las funciones miembro estáticas existen y pueden utilizarse incluso si no se crean instancias de los objetos de esa clase.

**17.15** El programador puede crear nuevos tipos a través del mecanismo de la clase. Estos nuevos tipos pueden designarse para utilizarlos de manera tan conveniente como los tipos predefinidos. Por lo tanto, C++ es un lenguaje extensible. Aunque el lenguaje es fácil de ampliar con estos nuevos tipos, el lenguaje base mismo no es modificable.

## *TIPS PARA PREVENIR ERRORES*

**17.1** Siempre declare las funciones miembro como **`const`**, si no modifican el objeto. Esto puede ayudar a eliminar errores.

**17.2** Los lenguajes como C++ son "blancos móviles" conforme evolucionan. Al lenguaje se adicionan más palabras reservadas. Evite utilizar palabras "cargadas", tales como "objeto", como identificadores. Aún cuando "objeto" no es una palabra reservada en C++, se podría convertir en una, de modo que la compilación con futuros compiladores podrían "romper" el código existente.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**17.1** Complete los espacios en blanco:

a) La sintaxis de ____________ se utiliza para inicializar los miembros constantes de una clase.
b) Una función no miembro debe declararse como ____________ de la clase para poder tener acceso a los datos miembro privados de la clase.
c) El operador ____________ asigna memoria de manera dinámica para un objeto de un tipo específico y devuelve un ____________ a ese tipo.
d) Un objeto constante debe ser ____________; una vez creado, no se puede modificar.
e) Un dato miembro ____________ representa información total de la clase.
f) Las funciones miembro de un objeto tienen acceso a un "autoapuntador" al objeto llamado el apuntador ____________.
g) La palabra reservada ____________ especifica que no puede modificarse un objeto o una variable una vez inicializada.
h) Si no se proporciona un inicializador de miembro para un objeto miembro de la clase, se llama al ____________ del objeto.
i) Una función miembro que se declara como **`static`** no puede acceder a los miembros ____________ de la clase.
j) Los objetos miembro se construyen ____________ antes que los objetos de clase que los encierran.
k) El operador ____________ reclama la memoria asignada previamente por **`new`**.

**17.2** Encuentre el error en cada una de las siguientes porciones de código y explique cómo corregirlo:

a)
```
class Ejemplo {
public:
   Ejemplo( int y = 10 ) { dato = y; }
   int obtieneDatoIncremento() const { return ++dato; }
   static int obtieneCuenta()
   {
      cout << "El dato es " << dato << endl;
      return cuenta;
    }
 private:
   int dato;
   static int cuenta;
};
```

b)
```
char *cadena;
cadena = new char[ 20 ];
free( cadena );
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**17.1** a) Inicialización de miembros. b) `friend`. c) `new`, apuntador. d) Inicializado. e) `static`. f) `this`. g) `const`. h) Constructor predeterminado. i) No estático. j) Antes. k) `delete`.

**17.2** a) Error: la definición de la clase para `Ejemplo` tiene dos errores. El primero ocurre en la función `obtieneDatoIncremento`. La función se declara como `const`, pero modifica al objeto.
Corrección: para corregir el primer error, elimine la palabra reservada `const` de la definición de `obtieneDatoIncremento`.
Error: el segundo error ocurre en la función `obtieneCuenta`. Esta función es estática, de modo que no se le permite el acceso a un miembro no estático de la clase.
Corrección: para corregir el segundo error, elimine la línea de salida de la definición de `obtieneCuenta`.
b) Error: la memoria asignada de manera dinámica por `new` se elimina con la función `free` de la biblioteca estándar de C.
Corrección: utilice el operador `delete` de C++ para reclamar la memoria, la asignación dinámica de memoria al estilo C no debe mezclarse con los operadores `new` y `delete` de C++.

## EJERCICIOS

**17.3** Compare y contraste la asignación dinámica de memoria por medio de los operadores `delete` y `new` de C++, con la asignación dinámica de memoria mediante las funciones `malloc` y `free` de la biblioteca estándar de C.

**17.4** Explique el concepto de amistad en C++. Explique los aspectos negativos de la amistad como los describe el libro.

**17.5** ¿Puede una definición correcta de la clase `Hora` incluir a ambos de los siguientes constructores? Si no es posible, explique por qué.

```
Hora( int h = 0, int m = 0, int s = 0 );
Hora();
```

**17.6** ¿Qué sucede si se especifica un tipo de retorno, incluso `void`, para un constructor o un destructor?

**17.7** Elabore una clase `Fecha` con las siguientes capacidades:
a) Despliegue la salida con múltiples formatos como:

```
DDD YYYY
MM/DD/YYYY
Junio 14, 1992
```

b) Utilice constructores sobrecargados para crear objetos `Fecha` inicializados con fechas en los formatos del inciso (a).
c) Elabore un constructor `Fecha` que lea la fecha del sistema mediante funciones de la biblioteca estándar del encabezado `<ctime>` y que establezca los miembros de `Fecha`.
En el capítulo 18, seremos capaces de crear operadores para evaluar la igualdad de dos fechas y de comparar fechas y determinar si una fecha es menor, o mayor que otra.

**17.8** Elabore la clase `CuentaAhorrros`. Utilice un dato miembro estático que contenga la `tasaInteresAnual` de cada uno de los ahorradores. Cada miembro de la clase debe contener un dato miembro privado `saldoAhorro` que indique el monto que el ahorrador tiene en depósito. Proporcione una función miembro `ultimoInteresMensual` que calcule el interés mensual al multiplicar el saldo por `tasaInteresAnual` dividida entre 12; este interés debe sumarse a `saldoAhorro`. Proporcione una función miembro estática `modificaTasaInteres` que establezca el nuevo valor de `tasaInteresAnual`. Escriba un programa principal que pruebe el funcionamiento de `CuentaAhorros`. Genere dos instancias de objetos `cuentaAhorros`, `ahorrador1`, `ahorrador2`, con saldos de `$2000.00` y `$3000.00`, respectivamente. Establezca `tasaInteresAnual` en 3%, luego calcule el interés mensual e imprima los nuevos saldos para cada uno de los ahorradores. Establezca `tasaInteresAnual` en 4% y calcule el interés del mes siguiente e imprima los nuevos saldos para cada uno de los ahorradores.

**17.9** Sería muy razonable que la clase `Hora` de la figura 17.8 representara internamente la hora como el número de segundos desde la medianoche y no cómo tres valores enteros para `hora`, `minuto`, `segundo`. Los clientes podrían utilizar los mismos métodos públicos y obtener los mismos resultados. Modifique la clase `Hora` de la figura 17.8 para implementar la clase `Hora` como el número de segundos desde medianoche y para mostrar que no existe un cambio visible en la funcionalidad de los clientes de la clase.

# 18

# Sobrecarga de operadores en C++

## Objetivos

- Comprender cómo redefinir (sobrecargar) operadores para trabajar con nuevos tipos.
- Comprender cómo convertir objetos de una clase a otra.
- Aprender cuándo sobrecargar operadores, y cuándo no hacerlo.
- Estudiar diversas clases interesantes que utilicen operadores sobrecargados.
- Crear una clase **`Arreglo`**.




*La total diferencia entre construcción y creación es exactamente ésta: que una cosa construida sólo puede ser amada después de que es construida; pero, una cosa creada es amada antes de que exista.*
Gilbert Keith Chesterton

*La muerte es casta.*
William Shakespeare

*Nuestro médico nunca operaría, a menos que fuera realmente necesario. Él simplemente fue así. Si no necesitara dinero, no le pondría una mano encima.*
Herb Shriner

## Plan general



## 18.1 Introducción

En los capítulos 16 y 17 presentamos los fundamentos de las clases de C++ y la idea de los tipos de datos abstractos (ADTs). Las manipulaciones sobre los objetos de clase (es decir, instancias de ADTs) se hicieron enviando mensajes (en la forma de llamadas a funciones miembro) a los objetos. Esta notación de llamadas a función es engorrosa para ciertos tipos de clases, en especial para las clases matemáticas. Para este tipo de clases sería bueno utilizar el rico conjunto de operadores de C++, para especificar las manipulaciones a objetos. En este capítulo, mostramos cómo permitir que los operadores de C++ trabajen con objetos de clase. A este proceso se le conoce como *sobrecarga de operadores*. Ampliar C++ con estas nuevas capacidades es directo y natural, sin embargo, también debe tener cuidado, ya que cuando no se hace buen uso de la sobrecarga, puede provocar que los programas sean difíciles de entender.

En C++, el operador `<<` tiene diversos propósitos; como operador de inserción de flujo, y como operador a nivel de bits de desplazamiento a la izquierda. Éste es un ejemplo de la sobrecarga de operadores. De manera similar, el operador `>>` también se sobrecarga; éste se utiliza como operador de extracción de flujo, y como operador a nivel de bits de desplazamiento a la derecha. Estos dos operadores están sobrecargados en la biblioteca de clases de C++. El lenguaje C++ mismo sobrecarga los operadores `+` y `-`. Estos operadores realizan diferentes tareas, de acuerdo con el contexto de la aritmética de enteros, de la aritmética de punto flotante o de la aritmética de apuntadores.

C++ permite al programador sobrecargar la mayoría de los operadores, para que sean sensibles al contexto en el que se utilizan. El compilador genera el código apropiado, basándose en la forma en la que se utiliza el operador. Algunos operadores se sobrecargan con frecuencia, en especial el operador de asignación y varios operadores aritméticos como `+` y `-`. El trabajo realizado por operadores sobrecargados también puede ser realizado por llamadas explícitas a funciones, pero la notación de operadores es más clara.

Explicaremos cuándo utilizar la sobrecarga de operadores y cuándo no. Mostraremos cómo sobrecargar operadores, y presentaremos muchos programas completos que utilizan operadores sobrecargados.

## 18.2 Fundamentos de la sobrecarga de operadores

La programación en C++ es sensible a los tipos y a los procesos que se enfocan en ellos. Los programadores pueden utilizar tipos integrados y puede definir nuevos tipos. Los tipos integrados pueden utilizarse con la rica colección de operadores de C++. Los operadores proporcionan a los programadores una notación concisa para expresar manipulaciones a objetos de tipos integrados.

Los programadores también pueden utilizar operadores con tipos definidos por el usuario. Aunque C++ no permite la creación de nuevos operadores, sí permite que la mayoría de los operadores existentes se sobrecarguen para que cuando se utilicen con objetos de clase, los operadores tengan un significado apropiado para los nuevos tipos. Ésta es una de las características más poderosas de C++.



**Observación de ingeniería de software 18.1**

*La sobrecarga de operadores contribuye a la extensibilidad de C++, uno de los atributos más atractivos del lenguaje.*



**Buena práctica de programación 18.1**

*Utilice la sobrecarga de operadores, cuando ésta haga que los programas sean más claros que si utilizara llamadas explícitas a funciones para realizar las mismas operaciones.*



**Buena práctica de programación 18.2**

*Evite el uso excesivo o inconsistente de la sobrecarga de operadores, ya que podría ocasionar que un programa fuera enigmático y difícil de leer.*

Aunque la sobrecarga de operadores puede sonar como una capacidad exótica, la mayoría de los programadores con frecuencia utilizan implícitamente operadores sobrecargados. Por ejemplo, el operador suma (**+**) funciona de manera muy diferente sobre enteros, flotantes y dobles. Sin embargo, la suma funciona bien con variables de tipo **int**, **float** y **double**, y un número de otros tipos integrados, ya que el operador suma (**+**) está sobrecargado en el lenguaje C++ mismo.

Los operadores se sobrecargan escribiendo una definición de función (con un encabezado y un cuerpo) como normalmente lo haría, con la excepción de que el nombre de la función ahora se convierte en la palabra reservada **operator**, seguida por el símbolo del operador que se está sobrecargando. Por ejemplo, el nombre de función **operator+** se utilizaría para sobrecargar el operador suma (**+**).

Para utilizar un operador sobre clases de objetos, ese operador *debe* sobrecargarse (existen dos excepciones). El operador de asignación (**=**) puede utilizarse con todas las clases, sin una sobrecarga explícita. El comportamiento predeterminado del operador de asignación es una *asignación de miembros* de los datos miembro de la clase. Pronto veremos que dicha asignación predeterminada de miembros es peligrosa para las clases con miembros apuntadores; explícitamente sobrecargaremos el operador de asignación para dichas clases. El operador de dirección (**&**) también puede utilizarse con objetos de cualquier clase sin tener que sobrecargarlos; éste simplemente devuelve la dirección del objeto en memoria. El operador de dirección también puede sobrecargarse.

La sobrecarga es más adecuada para clases matemáticas. Éstas con frecuencia requieren de un conjunto completo de operadores sobrecargados, para garantizar la consistencia con la forma en que se manejan realmente estas clases matemáticas. Por ejemplo, sería inusual sobrecargar sólo la suma para una clase de números complejos, ya que otros operadores aritméticos también se utilizan con frecuencia con dicho tipo de números.

C++ es un lenguaje rico en operadores. Es probable que los programadores en C++ que entienden el significado y contexto de cada operador hagan elecciones razonables, cuando se trata de sobrecargar operadores para clases nuevas.

El propósito de la sobrecarga de operadores es proporcionar las mismas expresiones concisas para tipos definidos por el usuario, que C++ proporciona en su rica colección de operadores para tipos integrados. Sin embargo, la sobrecarga de operadores no es automática; el programador debe escribir funciones para la sobrecarga de operadores, de tal modo que realicen las operaciones deseadas. Algunas veces, estas funciones se realizan mejor con funciones miembro; en ocasiones, son mejores como funciones **friend**, y en otras, pueden hacerse con funciones no miembro y no **friend**.

Es posible abusar en extremo de la sobrecarga, como en el caso del operador **+**, para realizar operaciones de tipo sustracción, o como en el caso del operador **/**, para realizar operaciones de tipo multiplicación. Tales usos de la sobrecarga hacen que un programa sea extremadamente difícil de entender.



**Buena práctica de programación 18.3**

*Sobrecargue operadores para que realicen la misma función o funciones similares sobre objetos de clase, que las que los operadores realizan sobre objetos de tipos integrados. Evite usos no intuitivos de los operadores.*

**Buena práctica de programación 18.4**

*Antes de escribir programas en C++ con operadores sobrecargados, consulte el manual de su compilador, para que tenga presentes las restricciones y requerimientos únicos de ciertos operadores en particular.*

## 18.3 Restricciones de la sobrecarga de los operadores

La mayoría de los operadores de C++ pueden sobrecargarse. Éstos aparecen en la figura 18.1. La figura 18.2 lista los operadores que no pueden sobrecargarse.



**Error común de programación 18.1**

*Intentar sobrecargar un operador que no puede sobrecargarse, es un error de sintaxis.*

La precedencia de un operador no puede modificarse por medio de la sobrecarga. Sobrecargar un operador cuya precedencia fija no es adecuada, puede propiciar situaciones confusas. Sin embargo, es posible utilizar paréntesis para forzar el orden de evaluación de los operadores sobrecargados de una expresión.

La asociatividad de un operador no puede modificarse por medio de la sobrecarga.

No es posible cambiar el número de operandos que toma un operador: los operadores unarios sobrecargados permanecen como operadores unarios; los operadores binarios sobrecargados permanecen como operadores binarios. El único operador ternario de C++ (**?:**) no puede sobrecargarse (figura 18.2). Los operadores **&**, *****, **+** y **-**, tienen versiones unarias y binarias; estas versiones pueden sobrecargarse separadamente.

No es posible crear nuevos operadores; sólo los operadores existentes pueden sobrecargarse. Desafortunadamente, esto evita que el programador utilice notaciones populares como la del operador ****** que se utiliza en FORTRAN y BASIC para la exponenciación.



**Error común de programación 18.2**

*Intentar crear nuevos operadores a través de la sobrecarga, es un error de sintaxis.*

No es posible modificar la manera en que un operador funciona con objetos de tipos integrados, por medio de la sobrecarga de operadores. Por ejemplo, el programador no puede modificar la manera en que **+** suma dos enteros. La sobrecarga de operadores sólo funciona con objetos de tipos definidos por el usuario o con una mezcla de un objeto de tipo definido por el usuario y un objeto de tipo integrado.



**Error común de programación 18.3**

*Intentar modificar la forma en que un operador funciona con objetos de tipos integrados, es un error de sintaxis.*

**Operadores que pueden sobrecargarse**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `+` | `-` | `*` | `/` | `%` | `^` | `&` | `\|` |
| `~` | `!` | `=` | `<` | `>` | `+=` | `-=` | `*=` |
| `/=` | `%=` | `^=` | `&=` | `\|=` | `<<` | `>>` | `>>=` |
| `<<=` | `==` | `!=` | `<=` | `>=` | `&&` | `\|\|` | `++` |
| `--` | `->*` | `,` | `->` | `[]` | `()` | `new` | `delete` |
| `new[]` | `delete[]` | | | | | | |

**Figura 18.1** Operadores que pueden sobrecargarse.

**Operadores que no pueden sobrecargarse**

| | | | | |
|---|---|---|---|---|
| `.` | `.*` | `::` | `?:` | `sizeof` |

**Figura 18.2** Operadores que no pueden sobrecargarse.



### Observación de ingeniería de software 18.2

*Al menos un argumento de una función operador debe ser un objeto de clase o una referencia a un objeto de clase. Esto evita que los programadores modifiquen la forma en que los operadores funcionan con tipos integrados.*

Sobrecargar un operador de asignación y un operador de suma para permitir instrucciones como

```
objeto2 = objeto2 + objeto1;
```

no implica que el operador += también se sobrecargue para permitir funciones como

```
objeto2 += objeto1;
```

Tal comportamiento puede lograrse sobrecargando explícitamente el operador **+=** para esa clase.



### Error común de programación 18.4

*Suponer que al sobrecargar un operador como* **+**, *se sobrecargan los operadores relacionados como* **+=**, *o que al sobrecargar el operador* **==**, *se sobrecarga un operador relacionado como* **!=**. *Los operadores pueden sobrecargarse solamente de manera explícita; no existe la sobrecarga implícita.*



### Error común de programación 18.5

*Intentar cambiar el número de operandos que toma un operador por medio de la sobrecarga, es un error de sintaxis.*



### Buena práctica de programación 18.5

*Para garantizar la consistencia entre operadores relacionados, utilice uno para implementar los otros (es decir, utilice un operador* **+** *sobrecargado, para implementar un operador* **+=** *sobrecargado).*

## 18.4 Funciones de operadores como miembros de una clase miembro *versus* funciones de operadores como funciones amigas (`friend`)

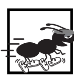
Las funciones operador pueden ser funciones miembro o funciones no miembro; las funciones no miembro a menudo se hacen amigas por razones de rendimiento. Las funciones miembro utilizan implícitamente el apuntador **this**, para obtener uno de los argumentos del objeto de la clase (el argumento izquierdo de los operadores binarios). En una llamada a una función miembro, ambos argumentos de la clase deben listarse explícitamente.

Cuando se sobrecargan los operadores **()**, **[]**, **->**, o cualquiera de los operadores de asignación, la función para sobrecargar al operador debe declararse como una clase miembro. Para los demás operadores, las funciones de sobrecarga pueden ser funciones no miembro.

Ya sea que una función operador se implemente como una función miembro o como una función no miembro, el operador se utiliza en las expresiones de la misma manera. Entonces, ¿cuál implementación es mejor?

Cuando una función operador se implementa como una función miembro, el operando más hacia la izquierda (o el único operando) debe ser un objeto de clase (o una referencia a un objeto de clase) correspondiente a la clase del operador. Si el operando izquierdo debe ser un objeto de una clase diferente o un tipo integrado, esta función operador debe implementarse como una función no miembro (tal y como haremos en la sección 18.5, cuando sobrecarguemos los operadores de inserción y extracción de flujo, **<<** y **>>**, respectivamente). Una función operador no miembro necesita ser amiga, si esa función debe acceder directamente a miembros privados o protegidos de esa clase.

El operador **<<** sobrecargado debe tener un operando izquierdo del tipo **ostream&** (tal como **cout** en la expresión **cout<<objetoClase**), por lo que debe ser una función no miembro. De manera similar, el operador **>>** sobrecargado debe tener un operando izquierdo del tipo **istream&** (tal como **cin** en la expresión **cin>>objetoClase**), por lo que éste, también debe ser una función no miembro. Además, cada una de estas funciones operador sobrecargadas pueden necesitar acceso a los datos miembro privados del objeto de clase que se está desplegando o introduciendo, por lo que en ocasiones se hace que estas funciones operador sobrecargadas sean amigas, por razones de rendimiento.



**Tip de rendimiento 18.1**

*Es posible sobrecargar un operador como una función no miembro y no amiga, pero una función como ésta, que necesita acceder a los datos privados o protegidos de una clase, necesitaría utilizar las funciones establecer u obtener provistas en la interfaz pública de esa clase. La sobrecarga producida por llamar a estas funciones podría ocasionar un rendimiento deficiente, por lo que se puede hacer que estas funciones sean* **`inline`** *para mejorar el rendimiento.*

Las funciones miembro correspondientes a una clase específica sólo se llaman cuando el operando izquierdo de un operador binario es específicamente un objeto de esa clase, o cuando el único operando de un operador unario es un objeto de esa clase.

Otra razón por la que uno puede elegir una función no miembro para sobrecargar un operador es permitir que el operador sea conmutativo. Por ejemplo, suponga que tenemos un objeto, **`numero`**, de tipo **`long int`**, y un objeto **`granEntero1`**, de clase **`EnteroEnorme`** (una clase en la que los enteros pueden ser arbitrariamente grandes, en lugar de que estén limitados por el tamaño de las palabras de la máquina del hardware adyacente; en los ejercicios de este capítulo desarrollamos la clase **`EnteroEnorme`**). El operador suma (**`+`**) produce un objeto temporal **`EnteroEnorme`** como la suma de un **`EnteroEnorme`** y un **`long int`** (como en la expresión **`granEntero1 + numero`**), o como la suma de un **`long int`** y un **`EnteroEnorme`** (como en la expresión **`numero + granEntero1`**). Entonces, requerimos que el operador suma sea conmutativo (como es normalmente). El problema es que el objeto de clase debe aparecer a la izquierda del operador suma, si ese operador va a sobrecargarse como una función miembro. Por lo tanto, sobrecargamos el operador como una función **`friend`** no miembro, para permitir que **`EnteroEnorme`** aparezca a la derecha de la suma. La función **`operator+`** que lidia con el **`EnteroEnorme`** a la izquierda aún puede ser una función miembro. Recuerde que una función no miembro no necesariamente tiene que ser amiga, si las funciones apropiadas *establecer* y *obtener* existen en la interfaz pública de la clase, y en especial, si las funciones establecer y obtener son **`inline`**.

## 18.5 Sobrecarga de los operadores de inserción y de extracción de flujo

C++ es capaz de introducir y desplegar los tipos de datos integrados a través de los operadores de inserción **`<<`** y de extracción **`>>`** de flujo. Estos operadores están sobrecargados (en la biblioteca de clases provista con los compiladores de C++) para procesar cada tipo de dato integrado, incluso cadenas, apuntadores y **`char *`** de estilo C. Los operadores de inserción y de extracción de flujo también pueden sobrecargarse para introducir y desplegar tipos de datos definidos por el usuario. La figura 18.3 muestra la sobrecarga de los operadores de inserción y de extracción de flujo para manejar datos de una clase definida por el usuario, llamada **`NumeroTelefonico`**. Este programa supone que los números telefónicos se introducen de manera correcta.

```
// Figura 18.3: fig18_03.cpp
// Sobrecarga de los operadores de inserción
// y de extracción de flujo.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

class NumeroTelefonico {
   friend ostream &operator<<( ostream&, const NumeroTelefonico & );
   friend istream &operator>>( istream&, NumeroTelefonico & );
```

**Figura 18.3** Operadores de inserción y de extracción de flujo definidos por el usuario. (Parte 1 de 2.)

```
19
20 private:
21    char codigoArea[ 4 ];   // código de área de tres dígitos y null
22    char intercambio[ 4 ];  // código de área de intercambio y  null
23    char linea[ 5 ];        // 4 dígitos para la línea y null
24 }; // fin de la clase NumeroTelefonico
25
26 // Sobrecarga el operador de inserción de flujo (no puede ser
27 // una función miembro si queremos invocarla con
28 // cout << algunNumeroTelefonico;).
29 ostream &operator<<( ostream &output, const NumeroTelefonico &num )
30 {
31    output << “(“ << num.codigoArea << “) “
32           << num.intercambio << “-” << num.linea;
33    return output;     // habilita cout << a << b << c;
34 } // fin de la función operator<<
35
36 istream &operator>>( istream &entrada, NumeroTelefonico &num )
37 {
38    entrada.ignore();                          // ignora (
39    entrada >> setw( 4 ) >> num.codigoArea;   // el código de área de entrada
40    entrada.ignore( 2 );                       // ignora ) y el espacio
41    entrada >> setw( 4 ) >> num.intercambio; // introduce intercambio
42    entrada.ignore();                          // ignora el guión (-)
43    entrada >> setw( 5 ) >> num.linea;        // introduce línea
44    return entrada;      // habilita cin >> a >> b >> c;
45 } // fin de la función operator>>
46
47 int main()
48 {
49    NumeroTelefonico telefono; // crea el objeto telefono
50
51    cout << “Introduzca un numero de telefono de la forma (123) 456-7890:\n”;
52
53    // cin >> telefono invoca a la función operator>> al
54    // ejecutar la llamada operator>>( cin, telefono ).
55    cin >> telefono;
56
57    // cout << telefono invoca a la función operator<< al
58    // ejecutar la llamada a operator<<( cout, telefono ).
59    cout << “El numero de telefono introducido fue: “ << telefono << endl;
60    return 0;
61 } // fin de la función main
```

```
Introduzca un numero de telefono de la forma (123) 456-7890:
(800) 555-1212
El numero de telefono introducido fue: (800) 555-1212
```

**Figura 18.3** Operadores de inserción y de extracción de flujo definidos por el usuario. (Parte 2 de 2.)

La función de operador de extracción de flujo **`operator>>`** (línea 36) toma como argumentos una referencia **`istream`** llamada **`introducir`** y una referencia a **`NumeroTelefonico`** llamada **`num`**, y devuelve una referencia **`istream`**. La función de operador **`operator>>`** se utiliza para introducir números telefónicos de la forma

```
(800) 555-1212
```

en objetos de la clase **NumeroTelefonico**. Cuando el compilador ve la expresión

```
cin >> telefono
```

en **main**, éste genera la llamada a función

```
operator>>( cin, telefono );
```

Cuando se ejecuta esta llamada, el parámetro de referencia introducir se vuelve un alias de **cin**, y el parámetro de referencia **num** se vuelve un alias de **telefono**. La función de operador lee como cadenas las tres partes del número telefónico que se encuentran en los miembros **codigoArea**, **intercambio** y **linea** del objeto referenciado **NumeroTelefonico** (**num** en la función operador y **telefono** en **main**). El manipulador de flujo **setw** restringe el número de caracteres leídos en cada arreglo de caracteres. Recuerde que, cuando se utiliza **cin**, **setw** restringe el número de caracteres leídos a uno menos que su argumento (es decir, **setw( 4 )** permite que se lean tres caracteres, y guarda una posición para el carácter de terminación nulo). Los caracteres correspondientes a paréntesis, espacios y guiones son evitados por medio de una llamada a la función miembro de **istream**, **ignore**, la cual descarta el número especificado de caracteres del flujo de entrada (de manera predeterminada, un carácter). La función **operator>>** devuelve la referencia de **istream**, **input**, es decir, **cin**. Esto permite que las operaciones de entrada a objetos **NumeroTelefonico** sean en cascada con las operaciones de entrada a otros objetos **NumeroTelefonico**, o a otros objetos de otros tipos de datos. Por ejemplo, podríamos introducir dos objetos **NumeroTelefonico** de la siguiente manera:

```
cin >> telefono1 >> telefono2;
```

Primero, la expresión **cin >> telefono1** se ejecutaría haciendo la llamada

```
operator>>( cin, telefono1 );
```

Esta llamada entonces devolvería una referencia a **cin** como el valor de **cin >> telefono1**, por lo que la parte restante de la expresión se interpretaría simplemente como **cin >> telefono2**. Ésta se ejecutaría haciendo la llamada

```
operator>>( cin, telefono2 );
```

El operador de inserción de flujo toma como argumentos una referencia **ostream** (**desplegar**) y una referencia (**num**) a un tipo definido por el usuario (**NumeroTelefonico**), y devuelve una referencia **ostream**. La función **operator<<** despliega objetos del tipo **NumeroTelefonico**. Cuando el compilador ve la expresión

```
cout << telefono
```

en **main**, éste genera la llamada a la función miembro

```
operator<<( cout, telefono );
```

La función **operator<<** despliega las partes del número telefónico en forma de cadenas, ya que están almacenadas en un formato de cadena.

Observe que las funciones **operator>>** y **operator<<** se declaran en la clase **NumeroTelefonico** como funciones amigas no miembros. Estos operadores deben ser no miembros, ya que el objeto de la clase **NumeroTelefonico** aparece, en cada caso, como el operando derecho del operador; para sobrecargar dicho operador como una función miembro, el operando de la clase debe aparecer a la izquierda del operador. Por razones de rendimiento, los operadores sobrecargados de inserción y de extracción se declaran como amigas, si necesitan acceder de manera directa a miembros no públicos de la clase. Además, observe que la referencia **NumeroTelefonico** de la lista de parámetros de **operator<<** es de tipo **const** (ya que el **NumeroTelefonico** simplemente se desplegará), y que la referencia **NumeroTelefonico** de la lista de parámetros de **operator>>** no es **const** (ya que el objeto **NumeroTelefonico** debe modificarse para almacenar en el objeto el número telefónico introducido).



**Observación de ingeniería de software 18.3**

*Es posible agregar a C++ nuevas capacidades de entrada/salida para tipos definidos por el usuario, sin modificar las declaraciones o los datos miembro **private** para cualquiera de las clases **ostream** o **istream**. Éste es otro ejemplo de la extensibilidad del lenguaje de programación C++.*

## 18.6 Sobrecarga de operadores unarios

Un operador unario para una clase puede sobrecargarse como una función miembro no **static** sin argumentos, o como una función no miembro con un argumento; ese argumento debe ser o un objeto de la clase, o una referencia a un objeto de la clase. Las funciones miembro que implementan operadores sobrecargados deben ser no estáticas, para que puedan acceder a los datos no estáticos de la clase. Recuerde que las funciones miembro estáticas sólo pueden acceder a datos miembro estáticos de la clase.

Podemos sobrecargar un operador unario **!** para evaluar si un objeto de la clase **Cadena** definida por el usuario está vacío, y devolver un resultado **bool**. Cuando se sobrecarga un operador unario como **!**, como una función miembro no estática sin argumentos, si **s** es un objeto de la clase **Cadena** o una referencia a un objeto de la clase **Cadena**, cuando el compilador ve la expresión **!s**, éste genera la llamada **s.operator!()**. El operando **s** es el objeto de la clase para el que se invoca a la función miembro de la clase **Cadena**, **operator!**. La función se declara en la definición de la clase de la siguiente manera:

```
class Cadena {
public:
   bool operator!() const;
   ...
}; // fin de la clase Cadena
```

Un operador unario como **!** puede sobrecargarse como una función no miembro con un argumento, de dos maneras: ya sea con un argumento que es un objeto (esto requiere una copia del objeto, para que los efectos colaterales de la función no se apliquen al objeto original), o con un argumento que sea una referencia a un objeto (no se hace copia alguna del objeto original, por lo que todos los efectos colaterales de esta función se aplican al objeto original). Si **s** es un objeto de la clase **Cadena** (o una referencia a un objeto de la clase **Cadena**), entonces **!s** se trata como si se hubiera escrito la llamada a **operator!( s )**, lo que provoca que se invoque a la función amiga no miembro de la clase **Cadena** que declaramos abajo:

```
class Cadena {
      friend bool operator!( const Cadena & );
      ...
 }; // fin de la clase Cadena
```



**Buena práctica de programación 18.6**

*Cuando se sobrecargan operadores unarios, es preferible hacer que las funciones operador sean miembros de la clase, en lugar de funciones amigas no miembros. Las funciones amigas y las clases amigas deben evitarse, a menos que sean absolutamente necesarias. Utilizar funciones amigas viola el encapsulamiento de una clase.*

## 18.7 Sobrecarga de operadores binarios

Un operador binario puede sobrecargarse como una función miembro no estática con un argumento, o como una función no miembro con dos argumentos (uno de esos argumentos debe ser un objeto de la clase o una referencia a un objeto de la clase).

Cuando se sobrecarga un operador binario como **+=,** como una función miembro no estática de la clase **Cadena** definida por el usuario con un argumento, si **y** y **z** son objetos de la clase **Cadena**, entonces **y += z** se trata como si se hubiera escrito **y.operator+=( z )**, lo que provoca que se invoque a la función miembro **operator+=** que declaramos abajo

```
class Cadena {
public:
   const Cadena &operator+=( const Cadena & );
   ...
}; // fin de la clase Cadena
```

Si va a sobrecargar el operador binario **+=** como una función no miembro, éste debe tomar dos argumentos; uno de los cuales debe ser un objeto de la clase o una referencia a un objeto de la clase. Si **y** y **z** son objetos de la clase **Cadena**, o referencias a objetos de la clase **Cadena**, entonces **y += z** se trata como si en el

programa se hubiera escrito la llamada **operator+=( y, z )**, lo que provoca que se invoque a la función amiga no miembro que declaramos abajo

```
class Cadena {
   friend const Cadena &operator+=( Cadena &, const Cadena & );
   ...
}; // fin de la clase Cadena
```

## 18.8 Ejemplo práctico: Una clase Arreglo

La notación de arreglos en C++ es sólo una alternativa a los apuntadores, por lo que los arreglos tienen mucha tendencia a errores. Por ejemplo, un programa puede fácilmente "tronar" a causa de un arreglo, ya que C++ no verifica si los subíndices se encuentran más allá del rango del arreglo. Los arreglos de tamaño *n* deben numerar sus elementos 0, ..., *n*-1; los rangos con subíndices alternados no están permitidos. Un arreglo de elementos que no son **char** no puede introducirse o desplegarse todo completo; cada elemento del arreglo debe leerse o escribirse de manera individual. Dos arreglos no pueden compararse significativamente con operadores de igualdad o con operadores de relación (ya que los nombres de los arreglos son simples apuntadores hacia el lugar donde los arreglos comienzan en memoria). Cuando un arreglo se pasa a una función de propósito general, diseñada para manejar arreglos de cualquier tamaño, el tamaño del arreglo debe pasarse como un argumento adicional. Un arreglo no puede asignarse a otro, por medio del operador de asignación (ya que los nombres de arreglo son apuntadores **const**, y un apuntador constante no puede utilizarse del lado izquierdo de un operador de asignación). Éstas y otras capacidades ciertamente parecen ser "naturales" para el manejo de arreglos, pero C++ no proporciona dichas capacidades. Sin embargo, C++ proporciona los medios para implementar dichas capacidades de arreglos, a través de los mecanismos de la sobrecarga de operadores.

En este ejemplo, desarrollamos una clase arreglo que realiza verificaciones de rango, para garantizar que los subíndices permanezcan dentro de los límites del arreglo. La clase permite que se asigne un arreglo a otro, por medio del operador de asignación. Los objetos de esta clase arreglo saben su tamaño, por lo que el tamaño no necesita pasarse como un argumento separado, cuando se pasa un arreglo a una función. Es posible introducir o desplegar arreglos completos por medio de los operadores de inserción y extracción de flujo, respectivamente. Las comparaciones de arreglos pueden realizarse con los operadores de igualdad **==** y **!=**. Nuestra clase arreglo utiliza un miembro **static** para dar seguimiento al número de objetos del arreglo que se han instanciado en el programa.

Este ejemplo mejorará su apreciación de la abstracción de datos. Probablemente usted querrá sugerir muchas mejoras a esta clase arreglo. El desarrollo de una clase es una actividad interesante, creativa e intelectualmente retadora; siempre con el objetivo de "crear clases valiosas".

El programa de la figura 18.4 muestra la clase **Arreglo** y sus operadores sobrecargados. Primero recorremos el programa principal en **main**. Después consideramos la definición de la clase y cada una de las definiciones de las funciones miembro de la clase y de las funciones **friend**.

```
// Figura 18.4: arreglo1.h
// Clase sencilla de Arreglo (para enteros)
#ifndef ARREGLO1_H
#define ARREGLO1_H

#include <iostream>

using std::ostream;
using std::istream;

class Arreglo {
   friend ostream &operator<<( ostream &, const Arreglo & );
   friend istream &operator>>( istream &, Arreglo & );
public:
```

**Figura 18.4** Una clase **Arreglo** con sobrecarga de operadores; **arreglo1.h**. (Parte 1 de 2.)

```
   Arreglo( int = 10 );                   // constructor predeterminado
   Arreglo( const Arreglo & );            // constructor de copia
   ~Arreglo();                            // destructor
   int obtenerTamanio() const;            // valor de retorno
   const Arreglo &operator=( const Arreglo & ); // asigna los arreglos
   bool operator==( const Arreglo & ) const;    // compara la igualdad

   // Determina si dos arreglos no son iguales y
   // devuelve true, de lo contrario devuelve false (utiliza operator==).
   bool operator!=( const Arreglo &derecha ) const
      { return ! ( *this == derecha ); }

   int &operator[]( int );                // operador de subíndice
   const int &operator[]( int ) const;    // operador de subíndice
   static int obtenerCuentaArreglo();     // Devuelve la cuenta de
                                          // los arreglos instanciados.
private:
   int tamanio; // tamanio del arreglo
   int *ptr; // apuntador al primer elemento del arreglo
   static int cuentaArreglo;  // # de Arreglos instanciados
}; // fin de la clase Arreglo

#endif
```

**Figura 18.4** Una clase **Arreglo** con sobrecarga de operadores; **arreglo1.h**. (Parte 2 de 2.)

```
// Figura 18.4: arreglo1.cpp
// Definición de las funciones miembro para la clase Arreglo
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>
#include <cassert>
#include “arreglo1.h”

// Inicializa el dato miembro static con alcance de archivo
int Arreglo::cuentaArreglo = 0;   // sin objetos aún

// Constructor predeterminado para la clase Arreglo (valor predeterminado de 10)
Arreglo::Arreglo( int tamanioArreglo )
{
   tamanio = ( tamanioArreglo > 0 ? tamanioArreglo : 10 );
   ptr = new int[ tamanio ];  // crea el espacio para el arreglo
   assert( ptr != 0 );        // termina si la memoria no está asignada
   ++cuentaArreglo;           // cuenta un objeto más

   for ( int i = 0; i < tamanio; i++ )
      ptr[ i ] = 0;           // inicializa el arreglo
```

**Figura 18.4** Una clase **Arreglo** con sobrecarga de operadores; **arreglo1.cpp**. (Parte 1 de 3.)

```
} // fin del constructor Arreglo

// El constructor de copia para la clase Arreglo
// debe recibir una referencia para prevenir una recursividad infinita
Arreglo::Arreglo( const Arreglo &init ) : tamanio( init.tamanio )
{
   ptr = new int[ tamanio ];  // crea el espacio para el arreglo
   assert( ptr != 0 );        // termina si la memoria no se asignó
   ++cuentaArreglo;           // cuenta un objeto más

   for ( int i = 0; i < tamanio; i++ )
      ptr[ i ] = init.ptr[ i ];  // copia init dentro del objeto
} // fin del constructor Arreglo

// Destructor para la clase Arreglo
Arreglo::~Arreglo()
{
   delete [] ptr;             // reclama espacio para el arreglo
   --cuentaArreglo;            // un objeto menos
} // fin del constructor Arreglo

// Obtiene el tamanio del arreglo
int Arreglo::obtenerTamanio() const { return tamanio; }

// Operador sobrecargado de asignación
// el retorno constante evita: ( a1 = a2 ) = a3
const Arreglo &Arreglo::operator=( const Arreglo &derecha )
{
   if ( &derecha != this ) {  // verifica la autoasignación

      // para arreglos de diferentes tamanios, desaloja el lado izquierdo
      // del arreglo original, luego desaloja el nuevo lado izquierdo del arreglo
      if ( tamanio != derecha.tamanio ) {
         delete [] ptr;             // reclama el espacio
         tamanio = derecha.tamanio; // modifica el tamaño de este objeto
         ptr = new int[ tamanio ];  // crea el espacio para la copia del arreglo
         assert( ptr != 0 );        // termina si no se asignó
      } // fin de if

      for ( int i = 0; i < tamanio; i++ )
         ptr[ i ] = derecha.ptr[ i ];  // copia el arreglo dentro del objeto
   } // fin de if

   return *this;   // permite x = y = z;
} // fin de la función operator=

// Determina si dos arreglos son iguales y
// devuelve true, de lo contrario devuelve false
bool Arreglo::operator==( const Arreglo &derecha ) const
{
   if ( tamanio != derecha.tamanio )
      return false;    // arreglos de diferentes tamaños

```

**Figura 18.4** Una clase **Arreglo** con sobrecarga de operadores; **arreglo1.cpp**. (Parte 2 de 3.)

```
   for ( int i = 0; i < tamanio; i++ )
      if ( ptr[ i ] != derecha.ptr[ i ] )
         return false; // los arreglos no son iguales

   return true;         // los arreglos son iguales
} // fin de la función operator==

// Operador subíndice sobrecargado para arreglos no constantes
// la referencia que devuelve crea un lvalue
int &Arreglo::operator[]( int subindice )
{
   // verifica si un subíndice se encuentra fuera de rango
   assert( 0 <= subindice && subindice < tamanio );

   return ptr[ subindice ]; // referencia devuelta
} // fin de la función operator[]

// Operador subíndice sobrecargado para arreglos constantes
// el retorno de la referencia constante crea un rvalue
const int &Arreglo::operator[]( int subindice ) const
{
   // verifica si un subíndice se encuentra fuera de rango
   assert( 0 <= subindice && subindice < tamanio );

   return ptr[ subindice ]; // referencia const devuelta
} // fin de la función operator[]

// Devuelve el número de objetos Arreglo instanciados
// las funciones estáticas no pueden ser const
int Arreglo::obtenerCuentaArreglo() { return cuentaArreglo; }

// Operador de entrada sobrecargado para la clase Arreglo;
// introduce valores para el arreglo completo.
istream &operator>>( istream &entrada, Arreglo &a )
{
   for ( int i = 0; i < a.tamanio; i++ )
      entrada >> a.ptr[ i ];

   return entrada;   // permite cin >> x >> y;
} // fin de la función operator>>

// Operador de salida sobrecargado para la clase Arreglo
ostream &operator<<( ostream &salida, const Arreglo &a )
{
   int i;

   for ( i = 0; i < a.tamanio; i++ ) {
      salida << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 números de salida por fila
         salida << endl;
   } // fin de for

   if ( i % 4 != 0 )
      salida << endl;

   return salida;   // permite cout << x << y;
} // fin de la funxión operator<<
```

**Figura 18.4** Una clase **`Arreglo`** con sobrecarga de operadores; **`arreglo1.cpp`**. (Parte 3 de 3.)

```
// Figura 18.4: fig18_04.cpp
// Controlador para una clase sencilla de Arreglo
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include “arreglo1.h”

int main()
{
   // aún no hay objetos
   cout << “# de arreglos instanciados = “
        << Arreglo::obtenerCuentaArreglo() << ‘\n’;

   // crea dos arreglos e imprime la cuenta de Arreglo
   Arreglo enteros1( 7 ), enteros2;
   cout << “# de arreglos instanciados = “
        << Arreglo::obtenerCuentaArreglo() << “\n\n”;

   // imprime el tamaño y el contenido de enteros1
   cout << “El tamanio del arreglo enteros1 es “
        << enteros1.obtenerTamanio()
        << “\nEl arreglo despues de la inicializacion:\n”
        << enteros1 << ‘\n’;

   // imprime el tamaño y el contenido de enteros2
   cout << “El tamanio del arreglo enteros2 es “
        << enteros2.obtenerTamanio()
        << “\nEl arreglo despues de la inicializacion:\n”
        << enteros2 << ‘\n’;

   // introduce e imprime enteros1 y enteros2
   cout << “Introduce 17 enteros:\n”;
   cin >> enteros1 >> enteros2;
   cout << “Despues de la entrada, los arreglos contienen:\n”
        << “enteros1:\n” << enteros1
        << “enteros2:\n” << enteros2 << ‘\n’;

   // utiliza el operador sobrecargado (!=)
   cout << “Evaluando: enteros1 != enteros2\n”;
   if ( enteros1 != enteros2 )
      cout << “No son iguales\n”;

   // crea el arreglo enteros3 con el uso de enteros1 como un
   // inicializador; imprime el tamaño y el contenido
   Arreglo enteros3( enteros1 );

   cout << “\nEl tamanio del arreglo enteros3 es “
         << enteros3.obtenerTamanio()
         << “\nEl arreglo despues de la inicializacion:\n”
         << enteros3 << ‘\n’;

   // utiliza el operador de asignación sobrecargado (=)
   cout << “Asigna enteros2 a enteros1:\n”;
   enteros1 = enteros2;
```

**Figura 18.4** Una clase **`Arreglo`** con sobrecarga de operadores; **`fig18_04.cpp`**. (Parte 1 de 3.)

```
   cout << "enteros1:\n" << enteros1
        << "enteros2:\n" << enteros2 << '\n';

   // utiliza el operador de igualdad sobrecargado (==)
   cout << "Evaluando: enteros1 == enteros2\n";
   if ( enteros1 == enteros2 )
      cout << "Son iguales\n\n";

   // utiliza el operador subíndice sobrecargado para crear un rvalue
   cout << "enteros1[5] is " << enteros1[ 5 ] << '\n';

   // utiliza el operador subíndice sobrecargado para crear un lvalue
   cout << "Asigna 1000 a enteros1[5]\n";
   enteros1[ 5 ] = 1000;
   cout << "enteros1:\n" << enteros1 << '\n';

   // intenta utilizar un subíndice fuera de rango
   cout << "Intenta asignar 1000 a enteros1[15]" << endl;
   enteros1[ 15 ] = 1000;  // ERROR: fuera de rango

   return 0;
} // fin de la función main
```

```
# de arreglos instanciados = 0
# de arreglos instanciados = 2

El tamanio del arreglo enteros1 es 7
El arreglo despues de la inicializacion:
            0            0            0            0
            0            0            0

El tamanio del arreglo enteros2 es 10
El arreglo despues de la inicializacion:
            0            0            0            0
            0            0            0            0
            0            0

Introduce 17 enteros:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Despues de la entrada, los arreglos contienen:
enteros1:
            1            2            3            4
            5            6            7
enteros2:
            8            9           10           11
           12           13           14           15
           16           17

Evaluando: enteros1 != enteros2
No son iguales

El tamanio del arreglo enteros3 es 7
El arreglo despues de la inicializacion:
            1            2            3            4
            5            6            7
```

**Figura 18.4** Una clase `Arreglo` con sobrecarga de operadores; `fig18_04.cpp`. (Parte 2 de 3.)

```
Asigna enteros2 a enteros1:
enteros1:
           8           9          10          11
          12          13          14          15
          16          17
enteros2:

           8           9          10          11
          12          13          14          15
          16          17

Evaluando: enteros1 == enteros2
Son iguales

enteros1[5] is 13
Asigna 1000 a enteros1[5]
enteros1:
           8           9          10          11
          12        1000          14          15
          16          17

Intenta asignar 1000 a enteros1[15]
Assertion failed: 0 <= subindice && subindice < tamanio, file C:\Documents and
Settings\Administrador\Mis documentos\Jorge Garcia\Cap18\FIG18_04\arreglo1.cpp,
line 95
```

**Figura 18.4** Una clase **`Arreglo`** con sobrecarga de operadores; **`fig18_04.cpp`**. (Parte 3 de 3.)

La variable estática **`cuentaArreglo`** de la clase **`Arreglo`** contiene el número de objetos en **`Arreglo`** que se instanciaron durante la ejecución del programa. El programa comienza utilizando a la función miembro estática **`obtenerCuentaArreglo`** (línea 192) para recuperar el número de arreglos instanciados hasta ese momento. Después, el programa crea instancias de dos objetos de la clase **`Arreglo`** (línea 195): **`enteros1`** con siete elementos y **`enteros2`**, cuyo tamaño predeterminado es de 10 elementos (el valor predeterminado que especificó el constructor predeterminado **`Arreglo`**). La línea 197 llama nuevamente a la función **`obtenerCuentaArreglo`** para recuperar el valor de la variable de clase **`cuentaArreglo`**. Las líneas 200 a 203 utilizan la función miembro **`obtenerTamanio`** para determinar el tamaño del **`Arreglo`** **`enteros1`** y desplegar **`enteros1`** por medio del operador de inserción de flujo sobrecargado **`Arreglo`**, para confirmar que el constructor inicializó correctamente los elementos del arreglo. Posteriormente, las líneas 206 a 209 despliegan el tamaño del arreglo **`enteros2`** y despliega **`enteros2`** por medio del operador de inserción de flujo sobrecargado **`Arreglo`**.

Después se le indica al usuario que introduzca 17 enteros. El operador de extracción de flujo sobrecargado **`Arreglo`** se utiliza para leer estos valores en ambos arreglos con la línea 213

```
cin >> enteros1 >> enteros2;
```

Los primeros siete valores se almacenan en **`enteros1`**, y los 10 valores restantes en **`enteros2`**. En las líneas 214 a 216, los dos arreglos se despliegan por medio del operador de inserción de flujo **`Arreglo`**, para confirmar que la entrada de dichos valores se llevó a cabo correctamente.

La línea 220 evalúa el operador de desigualdad sobrecargado, por medio de la condición

```
enteros1 != enteros2
```

y el programa reporta que los arreglos, de hecho, no son iguales.

La línea 225 crea una instancia de un tercer **`Arreglo`** llamado **`enteros3`**, y lo inicializa con el **`Arreglo`** **`enteros1`**. Esto provoca que se invoque al constructor de la copia de **`Arreglo`**, para copiar los elementos de **`enteros1`** en **`enteros3`**. En un momento, explicaremos los detalles del constructor de copia.

Las líneas 227 a 230 despliegan el tamaño de **enteros3** y despliegan **enteros3**, utilizando el operador de inserción de flujo sobrecargado **Arreglo** para confirmar que el constructor inicializó correctamente los elementos del arreglo.

Después, la línea 234 evalúa el operador de asignación (**=**) sobrecargado con la instrucción

```
enteros1 = enteros2;
```

Ambos **Arreglos** se imprimen en las líneas 235 y 236 para confirmar que la asignación fue exitosa. Observe que **enteros1** originalmente contenía 7 enteros, y que necesitaba modificar su tamaño para almacenar una copia de los 10 elementos de **enteros2**. Como veremos, el operador de asignación sobrecargado realiza esta modificación de tamaño de manera transparente para la función que invocó al operador.

Luego, la línea 240 utiliza el operador de igualdad (**==**) sobrecargado, para confirmar que los objetos **enteros1** y **enteros2** en realidad son idénticos después de la asignación.

La línea 244 utiliza el operador subíndice sobrecargado para hacer referencia a **enteros1[ 5 ]**; un elemento en el rango de **enteros1**. Este nombre con subíndice se utiliza como un *rvalue* en el lado izquierdo de una instrucción de asignación, para asignar un nuevo valor, 1000, al elemento 5 de **enteros1**. Observe que **operator[]** devuelve la referencia para utilizarla como el *lvalue*, después de que determina que **5** está en el rango de **enteros1**.

La línea 253 intenta asignar el valor **1000** a **enteros1[ 5 ]**; un elemento fuera de rango. El operador sobrecargado **[]** de **Arreglo** capta este error, y la ejecución del programa termina de manera anormal.

De manera interesante, el operador de subíndice del arreglo **[]** no está restringido únicamente a los arreglos; éste puede utilizarse para seleccionar elementos de otros tipos de clases contenedoras ordenadas como listas ligadas, cadenas, diccionarios, etcétera. Además, los subíndices ya no tienen que ser enteros; también pueden utilizarse caracteres, cadenas, números de punto flotante, e incluso objetos de clases definidas por el usuario.

Ahora que hemos visto cómo funciona el programa, veamos las definiciones del encabezado de la clase y de la función miembro. Las líneas 32 a 34

```
int tamanio; // tamaño del arreglo
int *ptr; // apuntador al primer elemento del arreglo
static int cuentaArreglo; // # de Arreglos instanciados
```

representan los datos miembros **private** de la clase. El arreglo consiste en un miembro **tamanio**, el cual indica el número de elementos del arreglo, un apuntador **int** (**ptr**), el cual apuntará al arreglo de enteros asignado dinámicamente y que está almacenado en un objeto **Arreglo**, y un miembro **static** (**arregloCuenta**), el cual indica el número de objetos del arreglo que se han instanciado.

Las línea 12 y 13

```
friend ostream &operator<<( ostream &, const Arreglo & );
friend istream &operator>>( istream &, Arreglo & );
```

declaran a los operadores de inserción y de extracción de flujo sobrecargados para que sean amigos de la clase **Arreglo**. Cuando el compilador ve una expresión como

```
cout << arregloObjeto
```

éste invoca a la función **operator<<( ostream &, const Arreglo & )**, generando la llamada

```
operator<<( cout, arregloObjeto )
```

Cuando el compilador ve una expresión como

```
cin >> arregloObjeto
```

éste invoca a la función **operator>>( istream &, Arreglo & )**, generando la llamada

```
operator>>( cin, arregloObjeto )
```

Nuevamente observamos que estas funciones de los operadores de inserción y de extracción de flujo no pueden ser miembros de la clase **Arreglo**, ya que el objeto **Arreglo** siempre se menciona del lado derecho de los operadores de inserción y de extracción de flujo. Si estas funciones de operador fueran miembros de la clase

**Arreglo**, las siguientes extrañas instrucciones tendrían que utilizarse para desplegar e introducir un **Arreglo**:

```
arregloObjeto << cout;
arregloObjeto >> cin;
```

La función **operator<<** (definida en la línea 162) imprime el número de elementos indicados por el tamaño del arreglo almacenado en **ptr**. La función **operator>>** (definido en la línea 153) introduce directamente en el arreglo apuntado por **ptr**. Cada una de estas funciones de operador devuelve una referencia apropiada, para permitir instrucciones de salida y de entrada en cascada, respectivamente.

La línea 15

```
Arreglo( int = 10 );          // constructor predeterminado
```

declara el constructor predeterminado para la clase, y especifica que el tamaño del arreglo se predetermina como de 10 elementos. Cuando el compilador ve una declaración como

```
Arreglo enteros1( 7 );
```

o la forma equivalente

```
Arreglo enteros1 = 7;
```

éste invoca al constructor predeterminado (recuerde que en este ejemplo, el constructor predeterminado recibe un solo argumento **int** que tiene un valor predeterminado de 10). El constructor predeterminado (definido en la línea 58) valida y asigna el argumento al dato miembro **tamanio**; utiliza **new** para obtener el espacio para almacenar la representación interna de este arreglo, y asigna el apuntador devuelto por **new** al dato miembro **ptr**; utiliza **assert** para evaluar si **new** estuvo bien; incrementa **cuentaArreglo**; y después utiliza un ciclo **for** para inicializar en cero todos los elementos del arreglo. Es posible tener una clase **Arreglo** que no inicialice sus miembros si, por ejemplo, estos miembros van a leerse más adelante. Sin embargo, esto se considera como una práctica de programación pobre. Los arreglos, y los objetos en general, deben mantenerse en todo momento en un estado consistente e incializados adecuadamente.

La línea 16

```
Arreglo( const Arreglo & );   // constructor de copia
```

declara un *constructor de copia* (definido en la línea 71) que inicializa un **Arreglo**, haciendo una copia de un objeto **Arreglo** existente. Dicha copia debe hacerse con cuidados para evitar el error de dejar ambos objetos **Arreglo** apuntando a la misma memoria asignada dinámicamente; ¡el problema que ocurriría con una copia de miembros predeterminada! Los constructores de copia se invocan siempre que sea necesaria una copia de un objeto, como en las llamadas por valor, cuando se devuelve por valor un objeto de una llamada a función, o cuando se inicializa un objeto que es la copia de otro objeto de la misma clase. El constructor de copia se llama en una definición, cuando se instancia y se inicializa un objeto de la clase **Arreglo** con otro objeto **Arreglo**, como en la siguiente declaración:

```
Arreglo enteros3( enteros1 );
```

o la declaración equivalente

```
Arreglo enteros3 = enteros1;
```

#### Error común de programación 18.6



*Observe que el constructor de copia debe utilizar una llamada por referencia, no una llamada por valor. De lo contrario, el constructor de copia puede dar como resultado una recursividad infinita (un error lógico fatal), ya que en una llamada por valor, se debe pasar una copia del objeto al constructor de copia, lo cual da como resultado ¡que se llame al constructor de copia de manera recursiva!*

El constructor de copia para **Arreglo** utiliza un inicializador miembro para copiar el tamaño del arreglo utilizado para la inicialización en el dato miembro **tamanio**; utiliza **new** para obtener el espacio para almacenar la representación interna de este arreglo y asigna el apuntador devuelto por **new** al dato miembro **ptr**; utiliza **assert** para evaluar que **new** estuvo bien; incrementa **cuentaArreglo**; y después utiliza un ciclo **for** para copiar todos los elementos del arreglo inicializador en este arreglo.



**Error común de programación 18.7**

*Si el constructor de copia simplemente copiara el apuntador del objeto fuente en el apuntador del objeto de interés, entonces ambos objetos apuntarían a la misma ubicación de memoria asignada dinámicamente. El primer destructor a ejecutarse entonces eliminaría la memoria asignada dinámicamente, y el otro apuntador del objeto entonces estaría indefinido; una situación conocida como apuntador indefinido, y con mucha probabilidad resultaría en un serio error de ejecución.*



**Observación de ingeniería de software 18.4**

*Un constructor, un destructor, un operador de asignación sobrecargado y un constructor de copia normalmente se proporcionan como grupo, para cualquier clase que utilice memoria asignada dinámicamente.*

La línea 17

```
~Arreglo();   // destructor
```

declara el destructor (definido en la línea 82) para la clase. El destructor se invoca cuando termina la vida de un objeto de la clase **`Arreglo`**. El destructor utiliza **`eliminar[]`** para solicitar el almacenamiento dinámico signado por nuevo en el constructor, después disminuye **`cuentaArreglo`**.

La línea 18

```
int obtenerTamanio() const; devuelve el tamaño
```

declara una función que lee el tamaño del arreglo.

La línea 19

```
const Arreglo &operator=( const Arreglo & );   // asigna arreglos
```

declara la función operador sobrecargada para la clase. Cuando el compilador ve una expresión como

```
enteros1 = enteros2;
```

éste invoca a la función **`operator=`**, generando la llamada

```
enteros1.operator=( enteros2 )
```

La función miembro **`operator=`** (definida en la línea 93) evalúa si se trata de una *autoasignación*. Si se intenta una autoasignación, la asignación se evita (es decir, el objeto ya es él mismo; en un momento veremos por qué es peligrosa la autoasignación). Si no se trata de una autoasignación, entonces la función miembro determina si los tamaños de los dos arreglos son idénticos, en cuyo caso, el arreglo original de enteros que se encuentra en el lado izquierdo del objeto **`Arreglo`**, no se reasigna. De lo contrario, **`operator=`** utiliza eliminar para solicitar el espacio originalmente asignado en el arreglo de destino; copia el tamaño del arreglo fuente en el tamaño del arreglo de destino; utiliza nuevo para asignar ese espacio para el arreglo de destino y coloca el apuntador devuelto por nuevo en el miembro **`ptr`** del arreglo; y utiliza **`assert`** para verificar que nuevo estuvo bien. Después, **`operator=`** utiliza un ciclo **`for`** para copiar los elementos del arreglo, desde el arreglo fuente hacia el arreglo de destino. Independientemente de que se trate de una autoasignación o no, la función miembro después devuelve el objeto actual (es decir, **`*this`**) como una referencia constante; esto permite asignaciones en cascada de **`Arreglo`**, como **`x = y = z`**.



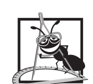
**Error común de programación 18.8**

*No proporcionar un operador de asignación sobrecargado y un constructor de copia para una clase, cuando los objetos de esa clase contienen apuntadores hacia memoria asignada dinámicamente, es un error lógico.*



**Observación de ingeniería de software 18.5**

*Es posible evitar que un objeto de una clase se asigne a otro. Esto se hace declarando al operador de asignación como un miembro privado de la clase.*



**Observación de ingeniería de software 18.6**

*Es posible evitar que los objetos de una clase se copien; para hacer esto, simplemente haga que tanto el operador de asignación sobrecargado como el constructor de copia sean privados.*

La línea 20

```
bool operator==(const Arreglo & ) const;   // compara la igualdad
```

declara al operador de igualdad sobrecargado (**==**) para la clase. Cuando el compilador ve la expresión

```
enteros1 == enteros2
```

en **main**, éste invoca a la función miembro **operator==**, generando la llamada

```
enteros1.operator==( enteros2 )
```

La función miembro **operator==** (definida en la línea 115) inmediatamente devuelve **false**, si los miembros **tamanio** de los arreglos son diferentes. De lo contrario, la función miembro compara cada par de elementos. Si éstos son los mismos, se devuelve **true**. El primer par de elementos que difieran ocasionará que se devuelva inmediatamente **false**.

Las líneas 24 y 25

```
bool operator!=( const Arreglo &derecha ) const
   { return ! ( *this == derecha ); }
```

define el operador de desigualdad (**!=**) sobrecargado para la clase. La función miembro **operator!=** se define en términos del operador de igualdad sobrecargado. La definición de la función utiliza la función **operator==** para determinar si un **Arreglo** es igual que otro; después devuelve el opuesto de ese resultado. Escribir la función **operator!=** de esta manera permite al programador reutilizar la función **operator==**, y reduce la cantidad de código que debe escribirse en la clase. Además, observe que toda la definición de la función **operator!=** se encuentra en el archivo de encabezado **Arreglo**. Esto permite al compilador hacer que la definición de **operator!=** sea **inline**, para eliminar la sobrecarga de llamadas adicionales a la función.

Las líneas 27 y 28

```
int &operator[]( int );               // operador de subíndice
const int &operator[]( int ) const;   // operador de subíndice
```

declaran dos operadores de subíndice sobrecargados (definidos en las líneas 129 y 139, respectivamente) para la clase. Cuando el compilador ve la expresión

```
enteros1[ 5 ]
```

en **main**, éste invoca a la función miembro sobrecargada **operator[]** apropiada, generando la llamada

```
enteros1.operator[]( 5 )
```

El compilador crea una llamada a la versión **const** de **operator[]**, cuando se utiliza el operador de subíndice sobre un objeto **const** de **Arreglo**. Por ejemplo, si se crea una instancia del objeto **const z** por medio de la instrucción

```
const Arreglo z( 5 );
```

después se requiere una versión **const** de **operator[]**, cuando una instrucción como

```
cout << z[ 3 ] << endl;
```

se ejecuta. Un objeto **const** sólo puede tener llamadas a sus funciones miembro **const**.

Cada definición de **operator[]** evalúa si el subíndice está en rango, y si no lo está, el programa termina de manera anormal. Si el subíndice está en rango, se devuelve el elemento apropiado del arreglo como una referencia, para que ésta pueda utilizarse como un *lvalue* (por ejemplo, en el lado izquierdo de una instrucción de asignación) en el caso de una versión no constante de **operator[]**, o como un *rvalue* en el caso de una versión constante de **operator[]**.

La línea 29

```
static int obtenerCuentaArreglos();   // devuelve la cuenta de Arreglos
```

declara como **static** la función **obtenerCuentaArreglos**, la cual devuelve el valor del dato miembro **static**, **cuentaArreglo**, incluso si no existen objetos de la clase **Arreglo**.

## 18.9 Conversión entre tipos

La mayoría de los programas procesan información de una variedad de tipos. Algunas veces las operaciones "permanecen de un tipo". Por ejemplo, sumar un entero con otro entero produce un entero (mientras el resultado

no sea demasiado grande como para representarlo como entero). Sin embargo, con frecuencia es necesario convertir los datos de un tipo en otro diferente. Esto puede ocurrir en asignaciones, en cálculos, en los pasos de valores a funciones y en valores devueltos por funciones. El compilador sabe cómo realizar ciertas conversiones entre tipos integrados. Los programadores pueden forzar las conversiones entre tipos integrados por medio de la conversión de tipo.

Pero, ¿qué sucede con los tipos definidos por el usuario? El compilador no puede saber cómo realizar conversiones entre tipos definidos por el usuario y tipos integrados. El programador debe especificar cómo deben ocurrir dichas conversiones. Tales conversiones pueden llevarse a cabo por medio de *constructores de conversión*; esto es, constructores de un solo argumento que devuelven objetos de otros tipos (incluso tipos integrados) en objetos de una clase en particular.

Un *operador de conversión* (también conocido como *operador de conversión de tipo*) puede utilizarse para convertir un objeto de una clase en un objeto de otra clase, o en un objeto de un tipo integrado. Dicho operador de conversión debe ser una función miembro no **static**; esta clase de operador de conversión no puede ser una función **friend**.

El prototipo de función

```
A::operator char *() const;
```

declara una función de operador de conversión de tipo sobrecargada, para crear un objeto temporal **char ***, fuera de un objeto de un tipo definido por el usuario. Una *función de operador de conversión de tipo* sobrecargada no especifica un tipo de retorno; el tipo de retorno es el tipo al que un objeto se está convirtiendo. Si **s** es un objeto de una clase, cuando el compilador ve la expresión **(char *)s**, éste genera la llamada **s.operator char *()**. El operando **s** es el objeto de la clase para el que la función miembro **operator char *()** se está invocando.

Las funciones de operador de conversión de tipo pueden definirse para convertir objetos de tipos definidos por el usuario en tipos integrados, o en objetos de otros tipos definidos por el usuario. Los prototipos

```
A::operator int() const;
A::operator otraClase() const;
```

declara las funciones operador de conversión de tipo sobrecargadas para convertir un objeto de un tipo definido por el usuario, **A**, en un entero, y para convertir un objeto de un tipo definido por el usuario, **A**, en un objeto de un tipo definido por el usuario, **otraClase**.

Una de las características buenas de los operadores de conversión de tipo y de los constructores de conversión es que, cuando es necesario, el compilador puede llamar estas funciones para crear objetos temporales. Por ejemplo, si un objeto **s** de una clase **Cadena** definida por el usuario aparece en un programa en una ubicación donde se espera un **char *** ordinario, como

```
cout << s;
```

el compilador llama a la función de operador de conversión de tipo sobrecargada **operator char *** de la expresión. Con este operador de conversión de tipo provisto por nuestra clase **Cadena**, el operador de inserción de flujo no necesita sobrecargarse para desplegar una **Cadena** por medio de **cout**.

## 18.10 Sobrecarga de ++ y --

Todos los operadores de incremento y decremento (preincremento, postincremento, predecremento y postdecremento) pueden sobrecargarse. Pronto veremos cómo es que el compilador distingue entre la versión prefija y la versión postfija de un operador de incremento o decremento.

Para sobrecargar el operador de incremento para permitir tanto el uso del operador de preincremento y postdecremento, cada función de operador sobrecargado debe tener una firma distinta para que el compilador sea capaz de determinar cuál versión de **++** se pretende. Las versiones prefijas se sobrecargan exactamente como cualquier otro prefijo de operador unario.

Por ejemplo, suponga que queremos sumar 1 al día **d1** del objeto **Fecha** definido por el usuario. Cuando el compilador ve la expresión de preincremento

```
++d1
```

el compilador genera la llamada a la función miembro

```
d1.operator++()
```

cuyo prototipo sería

```
Fecha &operator++();
```

Si el preincremento se implementa como una función no miembro, cuando el compilador ve la expresión

```
++d1
```

éste genera la llamada de función

```
operator++( d1 )
```

cuyo prototipo sería declarado en la clase **Fecha** como

```
friend Fecha &operator++( Fecha & );
```

Sobrecargar el operador de incremento representa un pequeño reto, ya que el compilador debe ser capaz de distinguir entre las firmas de las funciones de operador de preincremento y postincremento sobrecargadas. La convención que se ha adoptado en C++ es que cuando el compilador ve la expresión de postincremento

```
d1++
```

éste generará la llamada a la función miembro

```
d1.operator++( 0 )
```

cuyo prototipo es

```
Fecha operator++( int )
```

El **0** es estrictamente un "valor fantasma" para hacer que la lista de argumentos de **operator++**, utilizada para el postincremento, sea distinguible de la lista de argumentos de **operator++**, utilizada para el preincremento.

Si el postincremento se implementa como una función no miembro, cuando el compilador ve la expresión

```
d1++
```

el compilador genera la llamada de función

```
operator++( d1, 0 )
```

cuyo prototipo sería

```
friend Fecha operator++( Fecha &, int );
```

Una vez más, el compilador utiliza el argumento **0** para que la lista de argumentos de **operator++**, utilizada para el postincremento, sea distinguible de la lista de argumentos para el preincremento.

Todo lo que hemos explicado en esta sección para sobrecargar los operadores de preincremento y postincremento se aplica a la sobrecarga de los operadores de predecremento y postdecremento.

## RESUMEN

- En C++, el operador **<<** se utiliza con múltiples propósitos; como operador de inserción de flujo y como operador de desplazamiento a la izquierda. Éste es un ejemplo de la sobrecarga de operadores. De manera similar, **>>** también está sobrecargado; se utiliza tanto como operador de extracción de flujo y como operador de desplazamiento a la derecha.
- C++ permite al programador sobrecargar la mayoría de los operadores, para que sean sensibles al contexto en el que se utilizan. El compilador genera el código apropiado, basándose en el uso del operador.
- La sobrecarga de operadores contribuye a la extensibilidad de C++.
- Para sobrecargar un operador, escriba una definición de función; el nombre de la función debe ser la palabra reservada **operator**, seguido por el símbolo del operador que se está sobrecargando.

- Para utilizar un operador sobre objetos de una clase, ese operador *debe* sobrecargarse; existen dos excepciones. El operador de asignación (**=**) puede utilizarse con dos objetos de la misma clase para realizar una copia de miembros predeterminada, sin tener que sobrecargarlo. El operador de dirección (**&**) también puede utilizarse con objetos de cualquier clase, sin tener que sobrecargarlo; éste devuelve la dirección del objeto en memoria.
- La sobrecarga de operadores proporciona las mismas expresiones concisas para tipos definidos por el usuario que C++ proporciona con su rica colección de operadores que funcionan sobre tipos integrados.
- La precedencia y asociatividad de un operador no puede modificarse por medio de la sobrecarga.
- No es posible cambiar el número de operandos que toma un operador: los operadores unarios sobrecargados permanecen como operadores unarios; los operadores binarios sobrecargados permanecen como operadores binarios. El único operador ternario de C++, **?:**, no puede sobrecargarse.
- No es posible crear símbolos para operadores nuevos; sólo los operadores existentes pueden sobrecargarse.
- La forma como funciona un operador sobre tipos integrados, no puede modificarse mediante la sobrecarga.
- Cuando se sobrecargan los operadores **()**, **[]**, **->**, o cualquier operador de asignación, la función de sobrecarga de operador debe declararse como una clase miembro.
- Las funciones de operador pueden ser funciones miembro o no miembro.
- Cuando se implementa una función de operador como una función miembro, el operando más a la izquierda debe ser un objeto de la clase (o una referencia al objeto de la clase) correspondiente al operador.
- Si el operando izquierdo debe ser un objeto de una clase diferente, esta función de operador debe implementarse como una función no miembro.
- Las funciones miembro de operador se llaman sólo cuando el operando izquierdo de un operador binario es un objeto de esa clase, o cuando el único operando de un operador unario es un objeto de esa clase.
- Uno puede elegir una función no miembro para sobrecargar un operador, para que el operador sea conmutativo (es decir, dadas las definiciones adecuadas de un operador sobrecargado, el argumento izquierdo de un operador puede ser un objeto de otro tipo de dato).
- Un operador unario puede sobrecargarse como una función miembro no estática sin argumentos, o como una función miembro con un argumento; ese argumento debe ser un objeto de tipo definido por el usuario, o una referencia a un objeto de tipo definido por el usuario.
- Un operador binario puede sobrecargarse como una función miembro no estática con un argumento, o como una función no miembro con dos argumentos (uno de los cuales debe ser o un objeto de la clase, o una referencia un objeto de la clase).
- Un operador de subíndice **[]** no está restringido sólo para usarlo con arreglos; éste puede utilizarse para seleccionar elementos de otros tipos de clases contenedoras ordenadas, como listas ligadas, cadenas, diccionarios, etcétera. Además, los subíndices ya no tienen que ser enteros; por ejemplo, se podrían utilizar caracteres o cadenas.
- Un constructor de copia se utiliza para inicializar un objeto con otro objeto de la misma clase. Los constructores de copia también se invocan, siempre que la copia de un objeto se necesite, como en el caso de una llamada por valor y cuando se devuelve un valor desde la función llamada. En un constructor de copia, el objeto que se copia debe pasarse por referencia.
- El compilador no sabe cómo convertir entre tipos definidos por el usuario y tipos integrados; el programador debe especificar explícitamente cómo se van a realizar dichas conversiones. Tales conversiones pueden llevarse a cabo mediante constructores de conversión (es decir, constructores con un solo argumento) que simplemente cambian objetos de otros tipos en objetos de una clase en particular.
- Un operador de conversión (u operador de conversión de tipo) se utiliza para convertir un objeto de una clase en un objeto de otra, o en un objeto de un tipo integrado. Tales operadores de conversión deben ser funciones miembro no estáticas; esta clase de operadores de conversión, no pueden ser funciones amigas.
- Un constructor de conversión es un constructor con un solo argumento que se utiliza para convertir el argumento en un objeto de la clase del constructor. El compilador puede llamar implícitamente a dicho constructor.
- El operador de asignación es el operador que con mayor frecuencia se sobrecarga. Éste normalmente se utiliza para asignar un objeto a otro objeto de la misma clase, pero a través del uso de constructores de conversión, éste también puede utilizarse para asignaciones entre clases diferentes.
- Si no se define un operador de asignación sobrecargado, la asignación aún se permite, pero de manera predeterminada provoca la copia de los miembros. En algunos casos esto es aceptable. Para objetos que contienen apuntadores hacia memoria asignada dinámicamente, la copia de miembros da como resultado dos objetos que apuntan hacia esa misma memoria. Cuando se llama al destructor de cualquiera de estos objetos, se libera la memoria asignada dinámicamente. Si el otro objeto más adelante hace referencia a esa ubicación, el resultado es indefinido.

• Para sobrecargar al operador de incremento, para permitir el uso del preincremento y el postincremento, cada función de operador sobrecargada debe tener una firma diferente, de tal manera que el compilador sea capaz de determinar qué versión de ++ se pretende. Las versiones prefijas se sobrecargan como cualquier otro prefijo de operador unario. Es posible proporcionar una firma única para el operador de postincremento, proporcionando un segundo argumento, el cual debe ser de tipo **int**. De hecho, el usuario no proporciona un valor para este argumento entero especial. Éste tan solo sirve para ayudar al compilador a distinguir las versiones prefija y postfija de los operadores de incremento y decremento.

## *TERMINOLOGÍA*

apuntador indefinido
autoasignación
clase **Arreglo**
clase **Cadena**
clase **EnteroEnorme**
clase **Fecha**
clase **NumeroTelefonico**
concatenación de cadenas
constructor con un solo argumento
constructor de conversión
constructor de copia
conversión definida por el usuario
conversiones entre tipos de clases
conversiones entre tipos integrados y clases
conversiones explícitas de tipo
conversiones implícitas de tipo
copia predeterminada de miembros
función de conversión
función de operador de conversión de tipo
operador **--** sobrecargado
operador **!=** sobrecargado
operador **[]** sobrecargado
operador **+** sobrecargado
operador **++** sobrecargado
operador **+=** sobrecargado
operador **<** sobrecargado
operador **<<** sobrecargado
operador **<=** sobrecargado
operador **=** sobrecargado
operador **==** sobrecargado
operador **>** sobrecargado
operador **>>** sobrecargado
operador de asignación (**=**) sobrecargado
operador de conversión
operador sobrecargado de función miembro
operadores implementados como funciones
operadores que no pueden sobrecargarse
operadores que pueden sobrecargarse
operadores sobrecargados en cascada
**operator—**
**operator char ***
**operator int**
**operator!**
**operator!=**
**operator()**
**operator[]**
**operator+**
**operator++**
**operator++( int )**
**operator+=**
**operator<**
**operator<<**
**operator<=**
**operator=**
**operator==**
**operator>**
**operator>=**
**operator>>**
palabra reservada **operator**
sobrecarga
sobrecarga de la versión postfija de un operador unario
sobrecarga de la versión prefija de un operador unario
sobrecarga de operadores
sobrecarga de un operador binario
sobrecarga de un operador unario
tipo definido por el usuario

## *ERRORES COMUNES DE PROGRAMACIÓN*

**18.1** Intentar sobrecargar un operador que no puede sobrecargarse, es un error de sintaxis.

**18.2** Intentar crear nuevos operadores a través de la sobrecarga, es un error de sintaxis.

**18.3** Intentar modificar la forma en que un operador funciona con objetos de tipos integrados, es un error de sintaxis.

**18.4** Suponer que al sobrecargar un operador como **+**, se sobrecargan los operadores relacionados como **+=**, o que al sobrecargar el operador **==**, se sobrecarga un operador relacionado como **!=**. Los operadores pueden sobrecargarse solamente de manera explícita; no existe la sobrecarga implícita.

**18.5** Intentar cambiar el número de operandos que toma un operador por medio de la sobrecarga, es un error de sintaxis.

**18.6** Observe que el constructor de copia debe utilizar una llamada por referencia, no una llamada por valor. De lo contrario, el constructor de copia puede dar como resultado una recursividad infinita (un error lógico fatal), ya que en una llamada por valor, se debe pasar una copia del objeto al constructor de copia, lo cual da como resultado ¡que se llame al constructor de copia de manera recursiva!

**18.7** Si el constructor de copia simplemente copiara el apuntador del objeto fuente en el apuntador del objeto de interés, entonces ambos objetos apuntarían a la misma ubicación de memoria asignada dinámicamente. El primer destructor a ejecutarse entonces eliminaría la memoria asignada dinámicamente, y el otro apuntador del objeto entonces estaría indefinido; una situación conocida como apuntador indefinido, y con mucha probabilidad resultaría en un serio error de ejecución.

**18.8** No proporcionar un operador de asignación sobrecargado y un constructor de copia para una clase, cuando los objetos de esa clase contienen apuntadores hacia memoria asignada dinámicamente, es un error lógico.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**18.1** Utilice la sobrecarga de operadores, cuando ésta haga que los programas sean más claros que si utilizara llamadas explícitas a funciones para realizar las mismas operaciones.

**18.2** Evite el uso excesivo o inconsistente de la sobrecarga de operadores, ya que podría ocasionar que un programa fuera enigmático y difícil de leer.

**18.3** Sobrecargue operadores para que realicen la misma función o funciones similares sobre objetos de clase, que las que los operadores realizan sobre objetos de tipos integrados. Evite usos no intuitivos de los operadores.

**18.4** Antes de escribir programas en C++ con operadores sobrecargados, consulte el manual de su compilador, para que tenga presentes las restricciones y requerimientos únicos de ciertos operadores en particular.

**18.5** Para garantizar la consistencia entre operadores relacionados, utilice uno para implementar los otros (es decir, utilice un operador **+** sobrecargado, para implementar un operador **+=** sobrecargado).

**18.6** Cuando se sobrecargan operadores unarios, es preferible hacer que las funciones operador sean miembros de la clase, en lugar de funciones amigas no miembros. Las funciones amigas y las clases amigas deben evitarse, a menos que sean absolutamente necesarias. Utilizar funciones amigas viola el encapsulamiento de una clase.

## TIP DE RENDIMIENTO

**18.1** Es posible sobrecargar un operador como una función no miembro y no amiga, pero una función como ésta, que necesita acceder a los datos privados o protegido de una clase, necesitaría utilizar las funciones establecer u obtener provistas en la interfaz pública de esa clase. La sobrecarga producida por llamar a estas funciones podría ocasionar un rendimiento deficiente, por lo que se puede hacer que estas funciones sean **`inline`** para mejorar el rendimiento.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**18.1** La sobrecarga de operadores contribuye a la extensibilidad de C++, uno de los atributos más atractivos del lenguaje.

**18.2** Al menos un argumento de una función operador debe ser un objeto de clase o una referencia a un objeto de clase. Esto evita que los programadores modifiquen la forma en que los operadores funcionan con tipos integrados.

**18.3** Es posible agregar a C++ nuevas capacidades de entrada/salida para tipos definidos por el usuario, sin modificar las declaraciones o los datos miembro **`private`** para cualquiera de las clases **`ostream`** o **`istream`**. Éste es otro ejemplo de la extensibilidad del lenguaje de programación C++.

**18.4** Un constructor, un destructor, un operador de asignación sobrecargado y un constructor de copia normalmente se proporcionan como grupo, para cualquier clase que utilice memoria asignada dinámicamente.

**18.5** Es posible evitar que un objeto de una clase se asigne a otro. Esto se hace declarando al operador de asignación como un miembro privado de la clase.

**18.6** Es posible evitar que los objetos de una clase se copien; para hacer esto, simplemente haga que tanto el operador de asignación sobrecargado como el constructor de copia sean privado.

## EJERCICIOS DE AUTOEVALUACIÓN

**18.1** Complete los espacios en blanco:

a) Suponga que **`a`** y **`b`** son variables enteras y que formamos la suma **`a + b`**. Ahora suponga que **`c`** y **`d`** son variables de punto flotante y que formamos la suma **`c + d`**. Aquí, los dos operadores **+** claramente se están utilizando con propósitos diferentes. Éste es un ejemplo de la ____________.

b) La palabra reservada ____________ introduce una definición de función operador sobrecargada.

c) Para utilizar operadores sobre objetos de una clase, éstos deben sobrecargarse, con excepción de los operadores ____________ y ____________.

d) La ____________, la ____________ y el ____________ de un operador no pueden modificarse por medio de la sobrecarga.

**18.2** Explique los múltiples significados que los operadores **`<<`** y **`>>`** tienen en C++.

**18.3** ¿En qué contexto de C++ puede utilizarse el nombre **`operator/`**?

**18.4** (Verdadero/falso.) En C++, sólo los operadores existentes pueden sobrecargarse.

**18.5** En C++, ¿cómo resulta la comparación de la precedencia de un operador sobrecargado con la precedencia del operador original?

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**18.1** a) Sobrecarga de operadores. b) **`operator`**. c) Asignación (**`=`**), dirección (**`&`**). d) Precedencia, asociatividad, número de operandos.

**18.2** El operador **`>>`** es tanto el operador de desplazamiento a la derecha como el operador de extracción de flujo, de acuerdo con el contexto. El operador **`<<`** es tanto el operador de desplazamiento a la izquierda como el operador de inserción de flujo, de acuerdo con el contexto.

**18.3** Para la sobrecarga de operadores: éste sería el nombre de una función que proporcionaría una versión sobrecargada del operador **`/`**.

**18.4** Verdadero.

**18.5** Idéntica.

## *EJERCICIOS*

**18.6** Proporcione tantos ejemplos como le sea posible de la sobrecarga de operadores implícita en C++. Proporcione un ejemplo razonable de una situación en la que querría sobrecargar explícitamente un operador en C++.

**18.7** Los operadores de C++ que no pueden sobrecargarse son ________, ________, ________, ________ y ________.

**18.8** (*Proyecto*.) C++ es un lenguaje que evoluciona, y siempre hay lenguajes nuevos en desarrollo. ¿Cuáles operadores recomendaría para agregarlos a C++, o a un futuro lenguaje como C++, que soportara tanto la programación por procedimientos como la programación orientada a objetos? Escriba una justificación cuidadosa. Usted podría considerar el enviar su sugerencia al comité de ANSI C++, o al grupo de noticias **`comp.std.c++`**.

**18.9** Sobrecargue el operador de subíndices para devolver el elemento más grande de una colección, el segundo más grande, el tercero, etcétera.

**18.10** Considere la clase **`Complejo`** que aparece en la figura 18.5. La clase permite operaciones sobre *números complejos*. Éstos son números de la forma **`parteReal+parteImaginaria *i`**, donde ***`i`*** tiene el valor:

$$\sqrt{-1}$$

a) Modifique la clase para permitir la entrada y la impresión de números complejos, por medio de los operadores sobrecargados **`>>`** y **`<<`**, respectivamente (usted debe eliminar la función **`imprime`** de la clase).
b) Sobrecargue el operador de multiplicación para permitir la multiplicación de dos números complejos, como en álgebra.
c) Sobrecargue los operadores **`==`** y **`!=`** para permitir las comparaciones de números complejos.

```
// Figura 18.5: complejo1.h
// Definición de la clase Complejo
#ifndef COMPLEJO1_H
#define COMPLEJO1_H

class Complejo {
public:
   Complejo( double = 0.0, double = 0.0 );        // constructor
   Complejo operator+( const Complejo & ) const;  // suma
   Complejo operator-( const Complejo & ) const;  // resta
   const Complejo &operator=( const Complejo & ); // asignación
   void imprime() const;                          // salida
private:
   double real;        // parte real
   double imaginario;  // parte imaginaria
}; // fin de la clase Complejo

#endif
```

**Figura 18.5** Una clase de números complejos; **`complejo1.h`**.

```
// Figura 18.5: complejo1.cpp
// Definición de las funciones miembro para la clase Complejo
#include <iostream>

using std::cout;

#include “complejo1.h”

// Constructor
Complejo::Complejo( double r, double i )
   : real( r ), imaginario( i ) { }

// Operador sobrecargado de suma
Complejo Complejo::operator+( const Complejo &operando2 ) const
{
   return Complejo( real + operando2.real,
                    imaginario + operando2.imaginario );
} // fin de la función operator+

// Operador sobrecargado de resta
Complejo Complejo::operator-( const Complejo &operando2 ) const
{
   return Complejo( real - operando2.real,
                    imaginario - operando2.imaginario );
} // fin de la función operator-

// Operador sobrecargado =
const Complejo& Complejo::operator=( const Complejo &derecha )
{
   real = derecha.real;
   imaginario = derecha.imaginario;
   return *this;   // permite la cascada
} // fin de la función operator=

// Despliega un objeto Complejo de la forma: (a, b)
void Complejo::imprime() const
   { cout << ‘(‘ << real << “, “ << imaginario << ‘)’; }
```

**Figura 18.5** Una clase de números complejos; **`complejo1.cpp`**.

```
// Figura 18.5: fig18_05.cpp
// Controlador para la clase Complejo
#include <iostream>

using std::cout;
using std::endl;

#include “complejo1.h”

int main()
{
   Complejo x, y( 4.3, 8.2 ), z( 3.3, 1.1 );

```

**Figura 18.5** Una clase de números complejos; **`fig18_05.cpp`**. (Parte 1 de 2.)

```
    cout << “x: “;
    x.imprime();
    cout << “\ny: “;
    y.imprime();
    cout << “\nz: “;
    z.imprime();

    x = y + z;
    cout << “\n\nx = y + z:\n”;
    x.imprime();
    cout << “ = “;
    y.imprime();
    cout << “ + “;
    z.imprime();

    x = y - z;
    cout << “\n\nx = y - z:\n”;
    x.imprime();
    cout << “ = “;
    y.imprime();
    cout << “ - “;
    z.imprime();
    cout << endl;

    return 0;
} // fin de la función main
```

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

**Figura 18.5** Una clase de números complejos; **`fig18_05.cpp`**. (Parte 2 de 2.)

**18.11** El programa de la figura 18.3 contiene el comentario

```
// Operador sobrecargado de inserción de flujo (no puede ser
// una función miembro, si queremos invocarlo por medio de
// cout << algunNumeroTelefonico;)
```

De hecho, no puede ser una función miembro de la clase **`ostream`**, pero puede ser una función miembro de la clase **`NumeroTelefonico`**, si deseáramos invocarlo por medio de las siguientes:

```
algunNumeroTelefonico.operator<<( cout );
```

o

```
algunNumeroTelefonico << cout;
```

Rescriba el programa de la figura 18.3 con el operador sobrecargado de inserción de flujo, **`operator<<`**, como una función miembro, y pruebe las dos instrucciones anteriores para demostrar que funcionan.

# 19

# Herencia en C++

## Objetivos

- Crear nuevas clases a través de la herencia de clases existentes.
- Comprender la manera en que la herencia promueve la reutilización de software.
- Comprender los conceptos de clases base y clases derivadas.

*No digas que conoces a alguien por completo, hasta que dividas una herencia con él.*
Johann Kasper Lavater

*Este método es para definirse como el número de la clase de todas las clases similares a la clase dada.*
Bertrand Russell

*Una baraja de naipes se construyó como la más pura de las jerarquías, cada carta es superior para aquellas por debajo de ésta, e inferior para aquellas por arriba de ésta.*
Ely Culbertson

*Es bueno heredar una biblioteca, pero es mejor formar una.*
Augustine Birrell

*Toma lo más importante del libro de otros.*
William Shakespeare



## Plan general



## 19.1 Introducción

En éste y en el siguiente capítulo explicaremos dos de las más importantes capacidades de la programación orientada a objetos, la *herencia* y el *polimorfismo*. La herencia es una forma de reutilización de software en la cual, las nuevas clases se crean a partir de clases existentes al absorber sus atributos y comportamientos, y redefiniendo o embelleciéndolas con las capacidades que requieren las nuevas clases. La reutilización de software ahorra tiempo en el desarrollo del programa. La herencia promueve la reutilización de software comprobado, depurado y de alta calidad, con lo que reduce los problemas una vez que el sistema se hace funcional. Estas posibilidades son excitantes. El polimorfismo nos permite escribir programas de manera general para manipular una gran variedad de clases existentes y otras aún por especificar. La herencia y el polimorfismo son técnicas efectivas para manipular la complejidad del software.

Cuando se crea una nueva clase, en lugar de escribir por completo los nuevos datos miembro y las funciones miembro, el programador puede designar que la nueva clase va a *heredar* los datos miembros y las funciones miembro de una *clase base* definida previamente. A la nueva clase se le conoce como *clase derivada*. Cada clase derivada por sí misma se convierte en candidata a ser una clase base de una futura clase derivada. Mediante la *herencia simple*, una clase se deriva desde una clase base. Con la *herencia múltiple*, una clase se deriva de diversas (posiblemente no relacionadas) clases base. La herencia simple es directa, mostraremos varios ejemplos que le permitirán volverse competente en poco tiempo. La herencia múltiple es compleja y susceptible a errores, aquí explicaremos brevemente este útil tema y le aconsejamos tener cuidado y estudiar con más profundidad antes de utilizar esta poderosa capacidad.

Una clase derivada puede agregar datos y funciones miembro por su cuenta, de modo que una clase derivada puede ser más grande que su clase base. Una clase derivada es más específica que su clase base y representa a un grupo más pequeño de objetos. Con la herencia simple, la clase derivada comienza por ser, en esencia, la misma que la clase base. La fuerza real de la herencia proviene de la habilidad de definir en la clase derivada adiciones, reemplazos o refinamientos a las características heredadas de la clase base.

C++ ofrece tres tipos de herencia: pública, protegida y privada. En este capítulo nos concentraremos en la herencia pública y explicaremos brevemente los otros tres tipos. La segunda forma, la herencia privada, puede

utilizarse como una forma alternativa de composición. La tercera forma, la herencia protegida, es una adición relativamente reciente a C++ y rara vez se utiliza. Con la herencia publica, cada objeto de una clase derivada también puede tratarse como un objeto de la clase base de dicha clase derivada. Sin embargo, lo inverso no es verdad, los objetos de la clase base no son objetos de las clases derivadas de dicha clase base. Aprovecharemos esta relación "un objeto de la clase derivada *es un* objeto de la clase base" para llevar a cabo algunas manipulaciones interesantes. Por ejemplo, podemos relacionar una gran variedad de objetos diferentes relacionados a través de la herencia dentro objetos de la clase base de una lista ligada. Esto permite que una variedad de objetos se procesen de una manera general. Como veremos en el siguiente capítulo, esta capacidad, llamada polimorfismo, es la clave principal de la programación orientada a objetos.

En este capítulo, agregaremos una nueva forma de control de acceso a miembros, a saber, el acceso protegido (**`protected`**). Las clases derivadas y sus amigas tienen acceso a los miembros protegidos de la clase base, mientras que las funciones no amigas, no derivadas no lo tienen.

La experiencia en la construcción de sistemas de software indica que las partes importantes del código lidian con casos especiales íntimamente relacionados. En dichos sistemas es difícil ver todo el "panorama" debido a que el diseñador y el programador se preocupan por los casos especiales. La programación orientada a objetos proporciona diversas formas de "ver el bosque a través de los árboles", un proceso llamado *abstracción*.

Si un programa se carga con casos especiales muy relacionados, entonces será común ver instrucciones **`switch`** que diferencien los casos especiales y que proporcionen la lógica de procesamiento para lidiar con cada caso en particular. En el capítulo 20, mostraremos cómo utilizar la herencia y el polimorfismo para remplazar dicha lógica de **`switch`** por una lógica más simple.

Aquí diferenciaremos las *relaciones es un* y *tiene un*. *Es un* se refiere a la herencia. En una relación *es un*, un objeto del tipo de una clase derivada también puede tratarse como un objeto del tipo de una clase base. *Tiene un* es composición (vea la figura 17.4). En una relación *tiene un*, un objeto de la clase *tiene* como miembros uno o más objetos de otras clases.

Una clase derivada no tiene acceso a los miembros privados de su clase base; permitir esto violaría el encapsulamiento de la clase base. Sin embargo, una clase derivada tiene acceso a los miembros públicos y privados de su clase base. Los miembros de la clase base que no deben ser accesibles para la clase derivada mediante la herencia se declaran como privados en la clase base. Una clase derivada puede acceder a los miembros privados de la clase base solamente a través del acceso a funciones proporcionadas por las interfaces publicas y protegidas de la clase base.

Un problema con la herencia es que las clases derivadas pueden heredar las implementaciones de las funciones miembro públicas que no deseamos que tenga, o que no debe tener expresamente. Cuando la implementación de un miembro de la clase base no es apropiada para la clase derivada, dicho miembro puede redefinirse en la clase derivada mediante la implementación apropiada. En algunos casos, la herencia pública es simplemente inapropiada.

Quizá sea más excitante la idea de que las nuevas clases pueden heredar a partir de *bibliotecas de clases* existentes. Las empresas desarrollan sus propias bibliotecas de clases y pueden aprovechar otras bibliotecas disponibles alrededor del mundo. En algún momento, el software se construirá predominantemente a partir de *componentes estándares reutilizables*, tal como con frecuencia se construye el hardware en la actualidad. Esto ayudará a cumplir los retos de desarrollar el software más poderoso que necesitaremos en el futuro.

## 19.2 Herencia: Clases base y clases derivadas

A menudo, un objeto de una clase en realidad también "es un" objeto de otra clase. Ciertamente un rectángulo *es un* cuadrilátero (como lo es un cuadrado, un paralelogramo o un trapezoide). Así, se puede decir que la clase **`Rectangulo`** *hereda* de la clase **`Cuadrilatero`**. En este contexto, a la clase **`Cuadrilatero`** se le llama *clase base* y a la clase **`Rectangulo`** se le llama *clase derivada*. Un rectángulo es un tipo de cuadrilátero, pero es incorrecto decir que un cuadrilátero *es un* rectángulo (el cuadrilátero podría, por ejemplo, ser un paralelogramo). La figura 19.1 muestra varios ejemplos de herencia.

Otros lenguajes orientados a objetos tales como Smalltalk y Java utilizan terminología diferente: en la herencia, a la clase base se le llama *superclase* (la cual representa un superconjunto de objetos) y a la clase derivada se le llama *subclase* (la cual representa un subconjunto de objetos).

| Clase Base | Clases Derivadas |
|---|---|
| `Estudiante` | `EstudianteUniversitario` |
| | `EstudianteTitulado` |
| `Figura` | `Circulo` |
| | `Triangulo` |
| | `Rectangulo` |
| `Prestamo` | `PrestamoAutomovil` |
| | `PrestamoMejorarCasa` |
| | `PrestamoHipotecario` |
| `Empleado` | `EmpleadoDocente` |
| | `EmpleadoAdministrativo` |
| `Cuenta` | `CuentaCheques` |
| | `CuentaAhorros` |

**Figura 19.1** Algunos ejemplos sencillos de herencia.

Por lo general, la herencia produce clases derivadas con *más* características que sus clases base, de modo que los términos superclases y subclases pueden ser confusos; evitaremos estos términos. Los objetos de clases derivadas pueden considerarse como objetos de sus propias clases base; esto implica que existen más objetos asociados con las clases base y menos objetos asociados con las clases derivadas, así que es razonable llamar las clases base "superclases" y a las clases derivadas "subclases".

La herencia forma estructuras jerárquicas en forma de árboles. Una clase base existe en una relación jerárquica con sus clases derivadas. Una clase ciertamente puede existir por sí misma, pero es cuando se utiliza la clase con el mecanismo de herencia que la clase se convierte en una clase base que suministra los atributos y el comportamiento para otras clases, o en una clase derivada que hereda los atributos y comportamientos.

Desarrollemos una sencilla jerarquía de herencia. Una típica comunidad universitaria tiene miles de personas que son miembros de la comunidad. Estas personas pueden ser empleados, estudiantes y exalumnos. Los empleados pueden ser docentes o administrativos. Los docentes pueden ser administradores (tales como jefes de departamento o asesores) o maestros de la facultad. En la figura 19.2 mostramos la jerarquía de herencia. Observe que algunos docentes también imparten clases, de modo que tenemos que utilizar herencia múltiple para crear una clase llamada **`AdministradorMaestro`**. Con frecuencia, los estudiantes trabajan para sus universidades, y a menudo los empleados toman cursos, de modo que sería razonable utilizar la herencia múltiple para crear una clase llamada **`EmpleadoEstudiante`**.

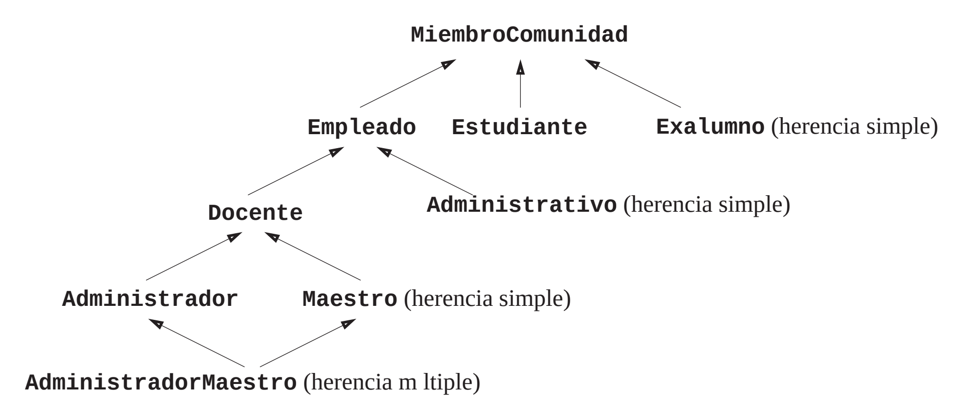



**Figura 19.2** Jerarquía de herencia para los miembros de la comunidad universitaria.

Otra jerarquía de herencia importante es la de **`Figura`**, la cual aparece en la figura 19.3. Una observación común entre los estudiantes que aprenden programación orientada a objetos es que en el mundo existen abundantes ejemplos de jerarquías. Solamente que estos estudiantes no están acostumbrados a clasificar el mundo de esta manera, por lo que es necesario hacer algunos ajustes en su manera de pensar.

Consideremos la sintaxis para indicar la herencia en una clase. Para especificar que la clase **`TrabajadorComision`** se deriva de la clase **`Empleado`**, por lo general, la clase **`TrabajadorComision`** se define de la siguiente manera:

```
Class TrabajadorComision : public Empleado {
   ...
}; // fin de la clase TrabajadorComision
```

A esto se le llama *herencia pública* y es el tipo de herencia más utilizada. También explicaremos la *herencia* privada y la herencia protegida. Con la herencia pública, los miembros públicos y protegidos de la clase base se heredan como miembros públicos y privados, respectivamente. Recuerde que los miembros privados de una clase base no están accesibles desde las clases derivadas de dicha clase. Observe que las funciones amigas no se heredan.

Es posible tratar a los objetos de clases base y a los objetos de clase derivadas de manera similar; esa similitud se expresa en los atributos y en el comportamiento de la clase. Los objetos de cualquier clase derivada mediante herencia pública de una clase base común pueden tratarse como objetos de la clase base. Veremos muchos ejemplos en los que podemos aprovechar esta relación con una programación sencilla no disponible en los lenguajes orientados a objetos, tales como C.

## 19.3 Miembros `protected`

Los miembros **`public`** de la clase base son accesibles para todas las funciones en el programa. Los miembros **`private`** de una clase base solamente son accesibles para las funciones miembro y **`friend`**s (amigas) de la clase base.

Introducimos el acceso **`protected`** como un nivel intermedio de protección entre el acceso público y el acceso privado. Se puede acceder a los miembros **`protected`** de la clase base solamente mediante miembros y amigas de la clase base, y por medio de los miembros y las amigas de la clase derivada. Los miembros de la clase derivada pueden hacer referencia a los miembros públicos y protegidos de la clase base simplemente utilizando los nombres de los miembros. Observe que los datos protegidos "rompen" el encapsulamiento; una modificación a los miembros **`protected`** de la clase base puede requerir la modificación de todas las clases derivadas.



**Observación de ingeniería de software 19.1**

*En general, declare los datos miembro de una clase como **`private`** y utilice **`protected`** solamente como "último recurso", cuando los sistemas necesiten cumplir ciertos requerimientos de rendimiento.*

## 19.4 Conversión de apuntadores de clases base en apuntadores de clases derivadas

Un objeto de una clase derivada pública también puede tratarse como un objeto de su clase base correspondiente. Esto hace posible algunas manipulaciones interesantes. Por ejemplo, no obstante el hecho de que los obje-

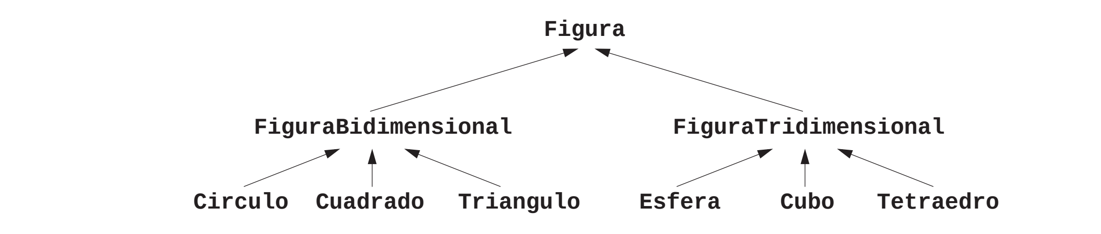



**Figura 19.3** Una parte de la jerarquía de clase de **`Figura`**.

tos de una variedad de clases se derivan de una clase base en particular pueden ser bastante diferentes uno de otro, podemos crear una lista ligada de ellos, de nuevo, mientras los tratemos como objetos de la clase base. Pero lo inverso no es verdad. Un objeto de la clase base no siempre es un objeto de la clase derivada.



**Error común de programación 19.1**

*Tratar un objeto de la clase base como un objeto de la clase derivada puede provocar errores.*

Sin embargo, el programador utiliza una conversión de tipo explícita para convertir el apuntador de una clase base a un apuntador de una clase derivada. Con frecuencia, este proceso es denominado *conversión hacia abajo de un apuntador*. Pero tenga cuidado, si dicho apuntador va a desreferenciarse, entonces el programador debe asegurarse de que el tipo del apuntador coincide con el tipo de objeto al cual apunta. En esta sección, nuestra explicación utiliza las técnicas ampliamente disponibles en la mayoría de los compiladores.



**Error común de programación 19.2**

*Convertir explícitamente un apuntador de una clase base que apunta a un objeto de la clase base en un apuntador de clase derivada, y después hacer referencia a los miembros de la clase derivada que no existen en dicho objeto, puede provocar errores lógicos en tiempo de ejecución.*

Nuestro primer ejemplo aparece en la figura 19.4. Las líneas 1 a 43 muestran la definición de la clase **`Punto`** y las definiciones de la función miembro **`Punto`**. Las líneas 44 a 106 muestran la definición de la clase **`Circulo`** y las definiciones de las funciones miembro de **`Circulo`**. Las líneas 107-147 muestran un programa controlador, en el cual demostramos cómo asignar apuntadores de una clase derivada a apuntadores de una clase base (con frecuencia llamada *conversión hacia arriba de un apuntador*) y cómo convertir apuntadores de la clase base en apuntadores de la clase derivada.

```
1  // Figura 19.4: punto.h
2  // Definición de la clase Punto
3  #ifndef PUNTO_H
4  #define PUNTO_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 class Punto {
11    friend ostream &operator<<( ostream &, const Punto & );
12 public:
13    Punto( int = 0, int = 0 );          // constructor predeterminado
14    void establecePunto( int, int );    // establece coordenadas
15    int obtieneX() const { return x; }  // obtiene la coordenada x
16    int obtieneY() const { return y; }  // obtiene la coordenada y
17 protected:          // accesible para las clases derivadas
18    int x, y;        // las coordenadas x y y de Punto
19 }; // fin de la clase Punto
20
21 #endif
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; **`punto.h`**.

```
22 // Figura 19.4: punto.cpp
23 // Funciones miembro para la clase Punto
24 #include <iostream>
25 #include "punto.h"
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; **`punto.cpp`**. (Parte 1 de 2.)

```

// Constructor para la clase Punto
Punto::Punto( int a, int b ) { establecePunto( a, b ); }

// Establece las coordenadas x y y de Punto
void Punto::establecePunto( int a, int b )
{
   x = a;
   y = b;
} // fin de la función establecePunto

// Despliega Punto (con el operador sobrecargado de inserción de flujo )
ostream &operator<<( ostream &salida, const Punto &p )
{
   salida << ‘[‘ << p.x << “, “ << p.y << ‘]’;

   return salida;   // permite llamadas en cascada
} // fin de la función operator<<
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; **`punto.cpp`**. (Parte 2 de 2.)

```
// Figura 19.4: circulo.h
// Definición de la clase Circulo
#ifndef CIRCULO_H
#define CIRCULO_H

#include <iostream>

using std::ostream;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include “punto.h”

class Circulo : public Punto {  // Circulo hereda de Punto
   friend ostream &operator<<( ostream &, const Circulo & );
public:
   // constructor predeterminado
   Circulo( double r = 0.0, int x = 0, int y = 0 );

   void estableceRadio( double );  // establece el radio
   double obtieneRadio() const;    // devuelve el radio
   double area() const;            // calcula el área
protected:
   double radio;
}; // fin de la clase Circulo

#endif
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; **`circulo.h`**.

```
// Figura 19.4: circulo.cpp
// Definición de las funciones miembro para la clase Circulo
#include “circulo.h”

// El constructor de Circulo llama al constructor de Punto
// mediante un inicializador de miembros y después inicializa el radio.
Circulo::Circulo( double r, int a, int b )
   : Punto( a, b )        // llama al constructor de la clase base
{ estableceRadio( r ); }

// Establece el radio del Circulo
void Circulo::estableceRadio( double r )
   { radio = ( r >= 0 ? r : 0 ); }

// Obtiene el radio del Circulo
double Circulo::obtieneRadio() const { return radio; }

// Calcula el área de Circulo
double Circulo::area() const
   { return 3.14159 * radio * radio; }

// Desliega un Circulo en la forma:
// Centro = [x, y]; Radio = #.##
ostream &operator<<( ostream &salida, const Circulo &c )
{
   salida << “Centro = “ << static_cast< Punto >( c )
          << “; Radio = “
          << setiosflags( ios::fixed | ios::showpoint )
          << setprecision( 2 ) << c.radio;

   return salida;   // permite llamadas en cascada
} // fin de la función operator<<
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; `circulo.cpp`.

```
// Figura 19.4: fig19_04.cpp
// Conversión de apuntadores de clases base en apuntadores de clases
   derivadas
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

#include “punto.h”
#include “circulo.h”

int main()
{
   Punto *ptrPunto = 0, p( 30, 50 );
   Circulo *ptrCirculo = 0, c( 2.7, 120, 89 );
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; `fig19_04.cpp`. (Parte 1 de 2.)

```

    cout << “Punto p: “ << p << “\nCirculo c: “ << c << ‘\n’;

    // Trata a Circulo como un Punto (solamente ve la parte de la clase base)
    ptrPunto = &c;   // asigna la dirección de Circulo a ptrPunto
    cout << “\nCirculo c (via *ptrPunto): “
         << *ptrPunto << ‘\n’;

    // Trata a Circulo como un Circulo (con alguna conversión)
    // convierte un apuntador de clase base en un apuntador de clase derivada
    ptrCirculo = static_cast< Circulo * >( ptrPunto );
    cout << “\nCirculo c (mediante *ptrCirculo):\n” << *ptrCirculo
         << “\nArea de c (mediante ptrCirculo): “
         << ptrCirculo->area() << ‘\n’;

    // PELIGRO: trata a un Punto como un Circulo
    ptrPunto = &p;   // asigna la dirección de Punto a ptrPunto

    // convierte el apuntador de clase base en un apuntador de clase derivada
    ptrCirculo = static_cast< Circulo * >( ptrPunto );
    cout << “\nPunto p (mediante *ptrCirculo):\n” << *ptrCirculo
         << “\nArea del objeto ptrCirculo apunta a: “
         << ptrCirculo->area() << endl;
    return 0;
} // fin de la función main
```

```
Punto p: [30, 50]
Circulo c: Centro = [120, 89]; Radio = 2.70

Circulo c (via *ptrPunto): [120, 89]

Circulo c (mediante *ptrCirculo):
Centro = [120, 89]; Radio = 2.70
Area de c (mediante ptrCirculo): 22.90

Punto p (mediante *ptrCirculo):
Centro = [30, 50]; Radio = 0.00
Area del objeto ptrCirculo apunta a: 0.00
```

**Figura 19.4** Conversión de apuntadores de la clase base en apuntadores de la clase derivada; **fig19_04.cpp**. (Parte 2 de 2.)

Examinemos la definición de la clase **Punto**. La interfaz pública de **Punto** incluye las funciones miembro **establecePunto**, **obtieneX** y **obtieneY**. Los datos miembro **x** y **y** de **Punto** se especifican como protegidos. Esto previene que los clientes de los objetos **Punto** accedan directamente a los datos, pero permite a las clases derivadas de **Punto** acceder directamente a los datos miembro heredados. Si los datos fueran privados, las funciones miembro públicas de **Punto** se utilizarían para acceder a los datos, incluso por las clases derivadas. Observe que la función sobrecargada del operador de inserción de flujo de **Punto** es capaz de hacer referencia a las variables **x** y **y** de manera directa, debido a que la función sobrecargada del operador de inserción de flujo es amiga de la clase **Punto**. Además, observe que es necesario hacer referencia a **x** y **y** a través de los objetos como en **p.x** y **p.y**. Esto se debe a que la función del operador de inserción de flujo no es una función miembro de la clase **Punto**, por lo que debemos utilizar un manipulador explícito para que el compilador sepa a cuál objeto hacemos referencia. Observe que esta clase ofrece las funciones miembro públicas **inline** **obtieneX** y **obtieneY**, así que **operator<<** no necesita ser una amiga para lograr un buen

rendimiento. Sin embargo, es posible que no se proporcionen las funciones miembro públicas necesarias en la interfaz pública de cada clase, por lo que con frecuencia la amistad es apropiada.

La clase **Circulo** hereda de la clase **Punto** mediante herencia pública. Esto se especifica en la primera línea de la definición de la clase:

```
Class Circulo : public Punto {   // Circulo hereda de Punto
```

Los dos puntos (**:**) en el encabezado de la definición de la clase indican la herencia. La palabra reservada **public** indica el tipo de herencia (en la sección 19.7 explicaremos la herencia protegida y privada). Todos los miembros públicos y protegidos de la clase **Punto** se heredan como miembros públicos y protegidos, respectivamente, dentro de la clase **Circulo**. Esto significa que la interfaz pública de **Circulo** incluye los miembros públicos de **Punto**, así como los miembros públicos de **Circulo**, **area**, **estableceRadio** y **obtieneRadio**.

El constructor **Circulo** debe invocar al constructor **Punto** para inicializar la porción de la clase base del objeto **Circulo**. Esto se lleva a cabo con un inicializador de miembros (introducido en el capítulo 17) de la siguiente manera:

```
Circulo::Circulo double r, int a, int b )
    : Punto( a, b )   // llama al constructor de la clase base
```

La segunda línea del encabezado del constructor invoca al constructor **Punto** por su nombre. Los valores **a** y **b** se pasan desde el constructor **Circulo** hasta el constructor **Punto** para inicializar a los miembros **x** y **y** de la clase base. Si el constructor **Circulo** no invoca al constructor **Punto** explícitamente, se invoca el constructor predeterminado de **Punto** de manera implícita con los valores predeterminados para **x** y **y** (es decir, 0 y 0). Si en este caso la clase **Punto** no proporcionó un constructor predeterminado, el compilador manda un mensaje de error de sintaxis. Observe que la función sobrecargada **operator<<** de **Circulo** es capaz de desplegar la parte **Punto** de **Circulo**, por medio de la conversión de la referencia **c** de **Circulo** a **Punto**. Esto genera una llamada a **operator<<** para **Punto** y despliega las coordenadas **x** y **y** utilizando el formato apropiado para **Punto**.

El programa controlador crea **ptrPunto** como un apuntador a un objeto **Punto** y crea la instancia del objeto **p** de **Punto**, luego crea **ptrCirculo** como un apuntador al objeto **Circulo** y crea la instancia del objeto **c** de **Circulo**. Los objetos **p** y **c** se despliegan por medio de sus operadores sobrecargados de inserción de flujo, para mostrar que se inicializaron correctamente. A continuación, el controlador asigna un apuntador a la clase derivada (la dirección del objeto **c**) para el apuntador de la clase base **ptrPunto**, y muestra el objeto **c** de **Circulo** mediante el uso de **operator<<** para **Punto** y el apuntador desreferenciado ***ptrPunto**. Observe que solamente se despliega la porción **Punto** del objeto **c** de **Circulo**. Con la herencia pública, siempre es válido asignar un apuntador de una clase derivada a un apuntador de la clase base, debido a que un objeto de la clase derivada *es un* objeto de la clase base. El apuntador de la clase base solamente "ve" la parte de la clase base del objeto de la clase derivada. El compilador realiza una conversión implícita del apuntador de la clase derivada en un apuntador de la clase base.

Luego, el programa controlador demuestra la conversión de **ptrPunto** de nuevo a **Circulo***. El resultado de la operación de conversión se asigna a **ptrCirculo**. El objeto **c** de **Circulo** se despliega con el uso del operador sobrecargado de inserción de flujo para **Circulo** y el apuntador desreferenciado ***ptrCirculo**. El **area** del objeto **c** de **Circulo** se despliega mediante **ptrCirculo**. Éste genera un área válida debido a que los apuntadores siempre apuntan a un objeto **Circulo**.

Un apuntador de una clase base no puede asignarse directamente a un apuntador de una clase derivada, debido a que ésta es una asignación peligrosa; los apuntadores de clases derivadas esperan apuntar a objetos de clases derivadas. En este caso, el compilador no realiza una conversión implícita. Por medio de una conversión explícita se informa al compilador que el programador sabe que este tipo de conversión de apuntador es peligrosa; el programador asume la responsabilidad de utilizar el apuntador de forma apropiada, así que el compilador puede permitir esta peligrosa conversión.

A continuación, el controlador asigna un apuntador de clase base (la dirección del objeto **p**) al apuntador **ptrPunto** de la clase base y realiza la conversión de **ptrPunto** de nuevo a **Circulo***. El resultado de la operación de conversión se asigna a **ptrCirculo**. El objeto **p** de **Punto** se despliega con el uso de

`operator<<` para `Circulo` y el apuntador desreferenciado `*ptrCirculo`. Observe el valor cero que se despliega para el miembro `radio` (el cual en realidad no existe, debido a que `ptrCirculo` apunta en realidad a un objeto `Punto`). Mostrar un `Punto` como un `Circulo` provoca un valor indefinido (en este caso sucede que es cero) para el `radio`, debido a que los apuntadores siempre apuntan a un objeto `Punto`. Un objeto `Punto` no tiene un miembro `radio`. Por lo tanto, el programa muestra cualquier valor que se encuentre en memoria en la posición en la que `ptrCirculo` espera se encuentre el dato miembro `radio`. El área del objeto al que apunta `ptrCirculo` (el objeto `p` de `Punto`) también se despliega mediante `ptrCirculo`. Observe que el valor para el área es `0.00` debido a que este cálculo se basa en el valor "indefinido" del `radio`. Obviamente, acceder a los datos miembro que no existen, es peligroso. Llamar a funciones miembro que no existen puede estropear el programa.

En esta sección mostramos la mecánica de la conversión de apuntadores. Este material establece los fundamentos que necesitaremos para tratar con más detalle a la programación orientada a objetos en el siguiente capítulo mediante el polimorfismo.

## 19.5 Uso de funciones miembro

Es posible que las funciones miembro de una clase derivada requieran tener acceso ciertos datos y funciones miembro de la clase base.



**Observación de ingeniería de software 19.2**

*Una clase derivada no puede acceder directamente a los miembros privados de su clase base.*

Éste es un aspecto crucial de la ingeniería de software en C++. Si una clase derivada pudiera acceder a los miembros privados de su clase base, esto violaría el encapsulamiento de la clase base. El ocultamiento de los miembros privados es una gran ayuda para la prueba, depuración y correcta modificación de los sistemas. Si una clase derivada pudiera acceder a los miembros privados de su clase base, entonces sería posible que las clases derivadas de dicha clase derivada también tuvieran acceso a esos datos, y así sucesivamente. Esto propagaría el acceso a lo que en teoría son datos privados, y se perderían los beneficios del encapsulamiento a través de la jerarquía de clases.

## 19.6 Cómo redefinir los miembros de una clase base en una clase derivada

Una clase derivada puede redefinir una función miembro de la clase base al suministrar una nueva versión de dicha función con la misma firma (si la firma fuera diferente, esto sería una sobrecarga de función y no una redefinición). Cuando se menciona a esa función por su nombre en la clase derivada, se selecciona la versión de la clase derivada. Se puede utilizar el operador de resolución de alcance para tener acceso a la versión de la clase base desde la clase derivada.

**Error común de programación 19.3**

*Cuando en una clase derivada se redefine una función miembro de la clase base, es común hacer que la versión de la clase derivada llame a la versión de la clase base y hacer algo de trabajo adicional. No utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base provoca una recursividad infinita, ya que la función miembro de la clase derivada en realidad se llama a sí misma. Esto provocará que en algún momento se agote la memoria del sistema; un error fatal en tiempo de ejecución.*

Considere la clase simplificada `Empleado`. Almacena el `nombre` y el `apellido` del empleado. Esta información es común para todos los empleados, incluso para las clases derivadas de la clase `Empleado`. A partir de la clase `Empleado` se derivan `EmpleadoXHora`, `EmpleadoXPieza`, `Jefe` y `EmpleadoXComision`. El `EmpleadoXHora` obtiene su pago por cada hora y recibe "una hora y media" por cada hora extra que excedan a las 40 horas semanales. El `empleadoXPieza` obtiene su pago mediante un pago fijo por pieza producida; por sencillez, asumimos que esta persona solamente hace un tipo de pieza, de modo que los datos miembro privados son el número de piezas producidas y el pago por pieza. El `Jefe` obtiene un salario fijo por semana. El `EmpleadoXComision` obtiene un pequeño salario fijo semanal más un porcentaje fijo de sus ventas totales por semana. Por sencillez, estudiamos solamente una clase `Empleado` y la clase derivada `EmpleadoXHora`.

```
// Figura 19.5: empleado.h
// Definición de la clase Empleado
#ifndef EMPLEADO_H
#define EMPLEADO_H

class Empleado {
public:
   Empleado( const char *, const char * );  // constructor
   void imprime() const;   // despliega el nombre y el apellido
   ~Empleado();            // destructor
private:
   char *nombre;         // cadena asignada dinámicamente
   char *apellido;       // cadena asignada dinámicamente
}; // fin de la clase Empleado

#endif
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`empleado.h`**.

```
// Figura 19.5: empleado.cpp
// Definición de las funciones miembro para la clase Empleado
#include <iostream>

using std::cout;

#include <cstring>
#include <cassert>
#include “empleado.h”

// El constructor asigna dinámicamente espacio para el
// nombre y el apellido, y utiliza strcpy para copiar
// el nombre y el apellido dentro del objeto.
Empleado::Empleado( const char *nomb, const char *apell )
{
   nombre = new char[ strlen( nomb ) + 1 ];
   assert( nombre != 0 );   // termina si no está permitido
   strcpy( nombre, nomb );

   apellido = new char[ strlen( apell ) + 1 ];
   assert( apellido != 0 ); // termina si no está permitido
   strcpy( apellido, apell );
} // fin del constructor Empleado

// Despliega el nombre del empleado
void Empleado::imprime() const
   { cout << nombre << ‘ ‘ << apellido; }

// El destructor libera la memoria asignada dinámicamente
Empleado::~Empleado()
{
   delete [] nombre;       // reclama la memoria dinámica
   delete [] apellido;     // reclama la memoria dinámica
} // fin del destructor Empleado
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`empleado.cpp`**.

```
// Figura 19.5: porHoras.h
// Definición de la clase EmpleadoXHoras
#ifndef PORHORAS_H
#define PORHORAS_H

#include "empleado.h"

class EmpleadoXHoras : public Empleado {
public:
   EmpleadoXHoras( const char*, const char*, double, double );
   double obtienePago() const;  // calcula y devuelve el salario
   void imprime() const;        // redefine imprime de la clase base
private:
   double pago;             // pago por horas
   double horas;            // horas trabajadas por semana
}; // fin de la clase EmpleadoXHoras

#endif
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`porHoras.h`**.

```
// Figura 19.5: porHoras.cpp
// Definición de las funciones miembro de la clase EmpleadoXHoras
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include "porHoras.h"

// Constructor para la clase EmpleadoXHoras
EmpleadoXHoras::EmpleadoXHoras( const char *primera,
                                const char *ultima,
                                double horasInic, double pagoInic )
   : Empleado( primera, ultima )   // llama al constructor de la clase base
{
   horas = horasInic;  // debe validarse
   pago = pagoInic;    // debe validarse
} // fin del constructor EmpleadoXHoras

// Obtiene el pago de EmpleadoXHoras
double EmpleadoXHoras::obtienePago() const { return pago * horas; }

// Imprime el nombre y el pago de EmpleadoXHoras
void EmpleadoXHoras::imprime() const
{
   cout << "EmpleadoXHoras::imprime() en ejecucion\n\n";
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`porHoras.cpp`**. (Parte 1 de 2.)

```
101      Empleado::imprime();    // llama a la función imprime de la clase base
102
103      cout << " es un empleado por horas con un pago de $"
104           << setiosflags( ios::fixed | ios::showpoint )
105           << setprecision( 2 ) << obtienePago() << endl;
106   } // fin de la función imprime
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`porHoras.cpp`**. (Parte 2 de 2.)

```
107   // Figura 19.5: fig.19_05.cpp
108   // Redefine una función miembro de la clase base en una
109   // clase derivada.
110   #include "porHoras.h"
111
112   int main()
113   {
114      EmpleadoXHoras h( "Juan", "Perez", 40.0, 10.00 );
115      h.imprime();
116      return 0;
117   } // fin de la función main
```

```
EmpleadoXHoras::imprime() en ejecucion

Juan Perez es un empleado por horas con un pago de $400.00
```

**Figura 19.5** Redefinición de miembros de la clase base en una clase derivada; **`fig19_05.cpp`**.

Mostramos nuestro siguiente ejemplo en la figura 19.5. Las líneas 1 a 50 muestran la definición de la clase **`Empleado`** y las definiciones de las funciones miembro de **`Empleado`**. Las líneas 51 a 106 muestran la definición de la clase **`EmpleadoXHora`** y la definición de la función miembro de **`EmpleadoXHora`**. Las líneas 107 a 117 muestran un programa controlador para la jerarquía de herencia **`Empleado/EmpleadoXHora`** que simplemente crea las instancias de un objeto **`EmpleadoXHora`**, lo inicializa y llama a la función miembro **`imprime`** de **`EmpleadoXHora`** para desplegar los datos del objeto.

La definición de la clase **`Empleado`** consiste en dos datos miembro privados **`char *`**, **`nombre`** y **`apellido`**, y tres funciones miembro, un constructor, un destructor e **`imprime`**. La función constructora recibe dos cadenas y asigna dinámicamente los arreglos de caracteres para almacenar las cadenas. Observe que utilizamos la macro **`assert`** para determinar si la memoria se asignó para almacenar el **`nombre`** o el **`apellido`**. Si no, el programa termina con un mensaje de error que indica la condición evaluada, el número de línea en la que aparece la condición y el archivo en el que se ubica la condición. [*Nota:* Una vez más, en el C++ estándar, **`new`** "lanza" una excepción si no hay suficiente memoria; esto lo explicaremos en el capítulo 23.] Los datos de **`Empleado`** son privados, de modo que el único acceso a los datos es a través de la función miembro **`imprime`**, la cual simplemente despliega el **`nombre`** y el **`apellido`** del empleado. La función destructora devuelve al sistema la memoria asignada dinámicamente (para evitar una "fuga de memoria").

La clase **`EmpleadoXHora`** hereda de la clase **`Empleado`** por medio de la herencia pública. De nuevo, esto se especifica en la primera línea de la definición de la clase, utilizando la notación de dos puntos (**`:`**), de la siguiente manera:

```
class EmpleadoXHora : public Empleado
```

La interfaz pública para **`EmpleadoXHora`** incluye la función **`imprime`** de **`Empleado`** y las funciones miembro **`obtienePago`** e **`imprime`** de **`EmpleadoXHora`**. Observe que **`EmpleadoXHora`** define su propia función **`imprime`** con el mismo prototipo que **`Empleado::imprime()`**; esto es un ejemplo de redefinición de

función. Por lo tanto, la clase **EmpleadoXHora** tiene acceso a dos funciones **imprime**. Además, la clase **EmpleadoXHora** contiene los datos miembro privados **pago** y **horas** para calcular el salario semanal.

El constructor **EmpleadoXHora** utiliza la sintaxis de inicialización de miembros para pasar las cadenas **primera** y **ultima** al constructor **Empleado** de modo que los miembros de la clase base puedan inicializarse, después inicializa los miembros **horas** y **pago**. La función miembro **obtienePago** calcula el salario de **EmpleadoXHora**.

La función miembro **imprime** de **EmpeladoXHora** redefine a la función miembro **imprime** de **Empleado**. Con frecuencia, las funciones miembro de la clase base se redefinen en la clase derivada para proporcionar más funcionalidad. Las funciones desplazadas con frecuencia llaman a la versión de la función de la clase base para realizar parte de la nueva tarea. En este ejemplo, la función **imprime** de la clase derivada llama a la función **imprime** de la clase base para desplegar la salida del nombre del empleado (la función **imprime** de la clase base es la única función con acceso a los datos privados de la clase base). La función **imprime** de la clase derivada también despliega el pago del empleado. Observe cómo se llama a la versión **imprime** de la clase base

```
Empleado::imprime();
```

La función de la clase base y la función de la clase derivada tiene el mismo nombre y firma, de modo que a la clase base debe antecederle su nombre de clase y el operador de resolución de alcance. De lo contrario, se podría llamar la versión de la clase derivada, ocasionando una recursividad infinita (es decir, la función **imprime** de **EmpleadoXHora** se llamaría a sí misma).

## 19.7 Herencia pública, protegida y privada

Cuando derivamos una clase a partir de una clase base, la clase base puede heredarse como pública, protegida o privada. El uso de la herencia protegida y privada es raro, y cada una debe utilizarse con mucho cuidado; por

| Especificador de acceso a miembros de la clase base | Tipo de herencia | | |
|---|---|---|---|
| | herencia **pública** | herencia **protegida** | herencia **privada** |
| **public** | **public** en una clase derivada.<br>Cualquier funci n miembro no est tica, funciones amigas y funciones no miembro pueden acceder directamente a ella. | **protected** en una clase derivada.<br>Funciones miembro no est ticas y funciones amigas pueden acceder directamente a ella. | **private** en una clase derivada.<br>Todas las funciones miembro y funciones amigas pueden acceder directamente a ella. |
| **protected** | **protected** en una clase derivada.<br>Todas las funciones miembro no est ticas y funciones amigas pueden acceder directamente a ella. | **protected** en una clase derivada.<br>Todas las funciones miembro no est ticas y funciones amigas pueden acceder directamente a ella. | **private** en una clase derivada.<br>Todas las funciones miembro no est ticas y funciones amigas pueden acceder directamente a ella. |
| **private** | Oculta en la clase derivada.<br>Se puede acceder a ella desde funciones miembro no est ticas y funciones amigas a trav s de funciones miembro p blicas o protegidas de la clase base. | Oculta en la clase derivada.<br>Se puede acceder a ella por medio de funciones miembro no est ticas y funciones amigas a trav s de funciones miembro p blicas o protegidas de la clase base. | Oculta en la clase derivada.<br>Se puede acceder a ella por medio de funciones miembro est ticas y funciones amigas a trav s de funciones miembro p blicas o protegidas de la clase base. |

**Figura 19.6** Resumen de la accesibilidad de miembros de la clase base en una clase derivada.

lo general, en este libro utilizamos la herencia pública. La figura 19.6 resume la accesibilidad de los miembros de la clase base desde la clase derivada para cada tipo de herencia. La primera columna contiene los especificadores de acceso a miembros de la clase base.

Cuando derivamos una clase desde una clase base pública, los miembros públicos de la clase base se hacen miembros públicos de la clase derivada, y los miembros protegidos de la clase base se hacen miembros protegidos de la clase derivada. Nunca se puede acceder a los miembros privados de una clase base desde la clase derivada, pero sí se puede acceder a ellos a través de llamadas a los miembros públicos y protegidos de la clase base.

Cuando derivamos desde una clase base protegida, los miembros públicos y protegidos de la clase base se hacen miembros protegidos de la clase derivada. Cuando derivamos desde la clase base privada, los miembros públicos y protegidos de la clase base se hacen miembros privados (por ejemplo, las funciones se vuelven funciones de utilidad) de la clase derivada. La herencia privada y protegida no son relaciones *es un*.

## 19.8 Clases base directas e indirectas

Una clase base puede ser una *clase base directa* de una clase derivada, o puede ser una *clase base indirecta* de la clase derivada. Una clase base directa de una clase derivada se lista explícitamente en el encabezado de la clase derivada con la notación de dos puntos (`:`), cuando se declara dicha clase derivada. Una clase base indirecta no se lista explícitamente en el encabezado de la clase derivada; en lugar de eso, la clase base se hereda desde dos o más niveles arriba en la jerarquía de clases.

## 19.9 Uso de constructores y destructores en clases derivadas

Una clase derivada hereda los miembros de su clase base, de modo que cuando se crea la instancia de un objeto de la clase derivada, es necesario llamar a cada constructor de la clase base para inicializar los miembros de la clase base del objeto de la clase derivada. Se puede proporcionar un *inicializador de la clase base* (el cual utiliza la sintaxis de inicialización de miembros que ya vimos) en el constructor de la clase derivada para llamar explícitamente al constructor de la clase base; de lo contrario, el constructor de la clase derivada llamará implícitamente al constructor predeterminado de la clase base.

Los constructores de la clase base y los operadores de asignación de la clase base no se heredan a las clases derivadas. Sin embargo, los constructores de la clase derivada y los operadores de asignación pueden llamar a los constructores de la clase base y a los operadores de asignación.

Un constructor de la clase derivada siempre llama primero al constructor su clase base para inicializar a los miembros de clase base correspondientes a la clase derivada. Si se omite el constructor de la clase derivada, el constructor predeterminado de la clase derivada llama al constructor predeterminado de la clase base. Los destructores se llaman en orden inverso al que se llama a los constructores, así que primero se llama al destructor de la clase derivada y después al destructor de la clase base.



**Observación de ingeniería de software 19.3**

*Suponga que creamos un objeto de una clase derivada, en donde tanto la clase base como la clase derivada contienen objetos de otras clases. Cuando se crea un objeto de dicha clase derivada, primero se ejecutan los constructores de los objetos miembros de la clase base, luego se ejecutan los constructores de la clase base, después se ejecutan los constructores de los objetos miembro de la clase derivada, y por último se llama a sus constructores correspondientes.*



**Observación de ingeniería de software 19.4**

*El orden en el cual se construyen los objetos miembro es el orden en el que se declaran dichos objetos dentro de la definición de la clase. El orden en el cual los inicializadores de miembros se listan no afecta el orden de construcción.*



**Observación de ingeniería de software 19.5**

*En la herencia, los constructores de la clase base se llaman en el orden en el que se especifica la herencia en la definición de la clase derivada. El orden en el cual se especifican los constructores de la clase base en la lista de inicialización de miembros de la clase derivada, no afecta el orden de la construcción.*

La figura 19.7 muestra el orden en el cual se llama a los constructores y a los destructores de la clase base y de la clase derivada. Las líneas 11 a 39 muestran una clase **Punto** sencilla que contiene un constructor, un destructor y los datos miembro protegidos **x** y **y**. Tanto el constructor como el destructor imprimen el objeto **Punto** para el que se invocaron.

```
1  // Figura 19.7: punto2.h
2  // Definición de la clase Punto
3  #ifndef PUNTO2_H
4  #define PUNTO2_H
5
6  class Punto {
7  public:
8     Punto( int = 0, int = 0 );  // constructor predeterminado
9     ~Punto();    // destructor
10 protected:      // accesible para las clases derivadas
11    int x, y;    // coordenadas x y y del Punto
12 }; // fin de la clase Punto
13
14 #endif
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **punto2.h**.

```
15 // Figura 19.7: punto2.cpp
16 // Definición de las funciones miembro de la clase Punto
17 #include <iostream>
18
19 using std::cout;
20 using std::endl;
21
22 #include “punto2.h”
23
24 // Constructor para la clase Punto
25 Punto::Punto( int a, int b )
26 {
27    x = a;
28    y = b;
29
30    cout << “constructor Punto: “
31         << ‘[‘ << x << “, “ << y << ‘]’ << endl;
32 } // fin del constructor Punto
33
34 // Destructor Punto
35 Punto::~Punto()
36 {
37    cout << “destructor Punto:  “
38         << ‘[‘ << x << “, “ << y << ‘]’ << endl;
39 } // fin del destructor Punto
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **punto2.cpp**.

```
// Figura 19.7: circulo2.h
// Definición de la clase Circulo
#ifndef CIRCULO2_H
#define CIRCULO2_H

#include “punto2.h”

class Circulo : public Punto {
public:
   // constructor predeterminado
   Circulo( double r = 0.0, int x = 0, int y = 0 );

   ~Circulo();
private:
   double radio;
}; // fin de la clase Circulo

#endif
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **`circulo2.h`**.

```
// Figura 19.7: circulo2.cpp
// Definición de las funciones miembro para la clase Circulo
#include <iostream>

using std::cout;
using std::endl;

#include “circulo2.h”

// El constructor para Circulo llama al constructor para Punto
Circulo::Circulo( double r, int a, int b )
   : Punto( a, b )   // llama al constructor de la clase base
{
   radio = r;  // debe validarse
   cout << “constructor Circulo: el radio es “
        << radio << “ [“ << x << “, “ << y << ‘]’ << endl;
} // fin del constructor Circulo

// Destructor para la clase Circulo
Circulo::~Circulo()
{
   cout << “destructor Circulo :  el radio es “
        << radio << “ [“ << x << “, “ << y << ‘]’ << endl;
} // fin del destructor Circulo
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **`circulo2.cpp`**.

```
// Figura 19.7: fig19_07.cpp
// Muestra cuándo se llama a los constructores y a los destructores
// de la clase base y de la clase derivada.
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **`fig19_07.cpp`**. (Parte 1 de 2.)

```
85  #include <iostream>
86
87  using std::cout;
88  using std::endl;
89
90  #include "punto2.h"
91  #include "circulo2.h"
92
93  int main()
94  {
95     // Muestra las llamadas al constructor y al destructor de la clase Punto
96     {
97        Punto p( 11, 22 );
98     } // fin del bloque
99
100    cout << endl;
101    Circulo circulo1( 4.5, 72, 29 );
102    cout << endl;
103    Circulo circulo2( 10, 5, 5 );
104    cout << endl;
105    return 0;
106 } // fin de la función main
```

```
constructor Punto: [11, 22]
destructor Punto:  [11, 22]

constructor Punto: [72, 29]
constructor Circulo: el radio es 4.5 [72, 29]

constructor Punto: [5, 5]
constructor Circulo: el radio es 10 [5, 5]

destructor Circulo :  el radio es 10 [5, 5]
destructor Punto:  [5, 5]
destructor Circulo :  el radio es 4.5 [72, 29]
destructor Punto:  [72, 29]
```

**Figura 19.7** Orden en el cual se invoca a los constructores y a los destructores de una clase base y de una clase derivada; **fig19_07.cpp**. (Parte 2 de 2.)

Las líneas 40 a 81 muestran una clase sencilla **Circulo** derivada de **Punto** con herencia pública. La clase **Circulo** proporciona un constructor, un destructor y un dato miembro privado llamado **radio**. Tanto el constructor como el destructor imprimen el objeto **Circulo** para el cual fueron invocados. El constructor **Círculo** también invoca al constructor **Punto** mediante el uso de la sintaxis de inicialización de miembros, y pasa los valores **a** y **b** de modo que los datos miembro **x** y **y** de la clase base puedan inicializarse.

Las líneas 82 a 106 son el programa controlador para esta jerarquía **Punto/Circulo**. El programa comienza con la creación de la instancia del objeto **Punto** con un alcance dentro de **main**. El objeto entra y sale de inmediato de alcance, así que tanto el constructor **Punto** como el destructor son invocados. A continuación, el programa crea la instancia **circulo1** del objeto **Circulo**. Esto invoca al constructor **Punto** para realizar la salida con valores pasados del constructor **Circulo**, y luego realiza la salida especificada en el constructor **Circulo**. A continuación, se crea la instancia del objeto **circulo2** de **Circulo**. De nuevo, se invocan los constructores **Punto** y **Circulo**. Observe que el cuerpo del constructor **Punto** se ejecuta antes del cuerpo del constructor **Circulo**. Se alcanza el final de **main**, de modo que se llama a los destructores para los objetos **circulo1** y **circulo2**. Los destructores se llaman en el orden inverso al de sus constructo-

res correspondientes. Por lo tanto, el destructor **`Circulo`** y el destructor **`Punto`** se llaman en ese orden para el objeto **`circulo2`**, después se llama a los destructores **`Circulo`** y **`Punto`**, en ese orden, para el objeto **`circulo1`**.

## 19.10 Conversión de objetos de clases derivadas a objetos de clases base

A pesar del hecho de que un objeto de clase derivada también *es un* objeto de la clase base, el tipo de la clase derivada y el tipo de la clase base son diferentes. En una herencia pública, los objetos de la clase derivada pueden tratarse como objetos de la clase base. Esto tiene sentido debido a que la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base; pero recuerde que la clase derivada puede tener más miembros que la clase base. La asignación en la otra dirección no está permitida, debido a que la asignación de un objeto de la clase base a un objeto de la clase derivada dejaría indefinidos los miembros adicionales de la clase derivada. Aunque dicha asignación no está permitida de modo "natural", podría hacerse legítima al proporcionar un operador de asignación sobrecargado apropiado y/o un constructor de conversión (vea el capítulo 18). Observe que lo que mencionamos acerca de los apuntadores en el resto de esta sección también se aplica a las referencias.



**Error común de programación 19.4**

*Asignar un objeto de clase derivada a un objeto de su clase base correspondiente, y luego intentar hacer referencia a miembros exclusivos de la clase derivada en el nuevo objeto de la clase base, es un error de sintaxis.*

Con la herencia pública, un apuntador a un objeto de clase derivada puede convertirse implícitamente en un apuntador de un objeto de clase base debido a que un objeto de clase derivada es un objeto de clase base.

Existen cuatro formas posibles de mezclar y de hacer coincidir apuntadores de clase base y apuntadores de clase derivada con objetos de clase base y objetos de clase derivada:

1. Hacer referencia a un objeto de la clase base con un apuntador de la clase base es directo.
2. Hacer referencia a un objeto de la clase derivada con un apuntador de la clase derivada es directo.
3. Hacer referencia a un objeto de clase derivada con un apuntador de la clase base es seguro, ya que el objeto de clase derivada también es un objeto de su clase base. Dicho código solamente puede hacer referencia a los miembros de la clase base. Si este código hace referencia a los miembros que son sólo de la clase derivada, a través del apuntador a la clase base, la computadora reportará un error de sintaxis.
4. Hacer referencia a un objeto de la clase base con un apuntador de la clase derivada, es un error de sintaxis. El apuntador de la clase derivada primero debe hacer la conversión a un apuntador de la clase base.



**Error común de programación 19.5**

*Convertir un apuntador de clase base en un apuntador de clase derivada puede provocar errores si dicho apuntador se utiliza para hacer referencia a un objeto de la clase base que no tiene los miembros requeridos en la clase derivada.*

Por conveniente que pueda ser tratar a los objetos de clases derivadas como objetos de clase base, y poder manipular todos estos objetos mediante apuntadores de clase base, existe un problema. Por ejemplo, en un sistema de nómina nos gustaría poder recorrer una lista ligada de empleados y calcular el pago semanal de cada persona. Pero el uso de los apuntadores de la clase base solamente permite al programa llamar a la rutina de cálculo de nómina de la clase base (si existiera dicha rutina en la clase base). Necesitamos una forma de invocar la rutina que calcule la nómina apropiada para cada objeto, ya sea un objeto de la clase base o un objeto de la clase derivada, y hacer esto simplemente con el uso del apuntador a la clase base. La solución es utilizar funciones virtuales y polimorfismo, como veremos en el capítulo 20.

## 19.11 Ingeniería de software con herencia

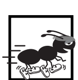
Podemos utilizar la herencia para personalizar el software existente. Heredamos los atributos y el comportamiento (o redefinimos el comportamiento de la clase base) para personalizar la clase de acuerdo con nuestras

necesidades. En C++, esto se hace sin que la clase derivada tenga acceso al código fuente de la clase base, pero la clase derivada necesita ser capaz de enlazarse al código del objeto de la clase base. Esta poderosa capacidad es atractiva para los fabricantes independientes de software. Dichos fabricantes pueden desarrollar clases propietarias para venta o licencia y pueden poner dichas clases a disposición de los usuarios con el formato de código objeto. Los usuarios pueden entonces derivar nuevas clases rápidamente desde esta biblioteca de clases sin acceder al código fuente propietario del fabricante. Todos los fabricantes independientes de software necesitan proporcionar los archivos de encabezado junto con el código objeto.



**Observación de ingeniería de software 19.6**

*En teoría, los usuarios no necesitan ver el código fuente de las clases de las cuales heredan. En la práctica, la gente que vende licencias de las clases nos ha dicho que con frecuencia los clientes requieren el código fuente. Al parecer, los programadores son reticentes a incorporar código dentro de sus programas cuando este código fue escrito por otras personas.*

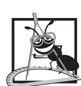


**Tip de rendimiento 19.1**

*Cuando el rendimiento es un asunto de mayor importancia, es posible que los programadores deseen ver el código fuente de las clases de las que heredan, de modo que puedan poner a punto el código para que cumpla con sus requerimientos de rendimiento.*

Para los estudiantes puede ser difícil apreciar el problema que enfrentan los diseñadores y los implementadores de proyectos de software a gran escala. La gente con experiencia en dichos proyectos invariablemente dirá que la clave para mejorar el proceso de desarrollo de software es la reutilización de software. En general la programación orientada a objetos, y en particular C++, ciertamente hacen esto.

La disponibilidad de bibliotecas de clases útiles y completas proporciona el máximo beneficio de la reutilización de software a través de la herencia. Al crecer el interés por C++, el interés por las bibliotecas de clases crece de manera exponencial. Tal como el software producido por fabricantes independientes de software tuvo un crecimiento explosivo en la industria con el arribo de la computadora personal, lo mismo sucede con la creación y venta de bibliotecas de clases. Los diseñadores de aplicaciones construyen sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas se ven recompensados al tener sus bibliotecas incluidas en sus aplicaciones. Las bibliotecas que se distribuyen con los compiladores de C++ tienden a ser de propósito general y de alcance limitado. En la actualidad existe un compromiso mundial para desarrollar bibliotecas de clases para una gran variedad de escenarios de aplicación.



**Observación de ingeniería de software 19.7**

*La creación de una clase derivada no afecta el código fuente o el código objeto de su clase base; la integridad de la clase base se preserva mediante la herencia.*

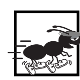
Una clase base especifica similitudes; todas las clases derivadas de la clase base heredan las capacidades de dicha clase base. En el proceso de diseño orientado a objetos, el diseñador busca las similitudes y las aprovecha para formar clases base apropiadas. Las clases derivadas entonces se personalizan más allá de las capacidades heredadas de la clase base.



**Observación de ingeniería de software 19.8**

*En un sistema orientado a objetos, con frecuencia las clases están íntimamente relacionadas. "Descubra" los atributos y los comportamientos comunes y colóquelos en una clase. Después utilice la herencia para formar clases derivadas.*

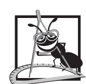
Tal como un diseñador de sistemas no orientados a objetos busca evitar la proliferación de funciones innecesarias, el diseñador de sistemas orientados a objetos debe evitar la proliferación de clases innecesarias. Tal proliferación de clases crea problemas de administración y puede dificultar la reutilización de software, simplemente debido a que es más difícil para un potencial reutilizador de esa clase localizar dicha clase dentro de una gran colección. El equilibrio se encuentra al crear menos clases, cada una con gran funcionalidad adicional. Dichas clases podrían ser demasiado grandes para ciertos usuarios; estos usuarios pueden disfrazar la funcionalidad excesiva, y así "aterrizar" las clases para ajustarlas a sus necesidades.



**Tip de rendimiento 19.2**

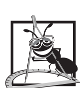
*Si las clases producidas a través de la herencia son más grandes de lo necesario, los recursos de memoria y programación pueden desperdiciarse. Herede de la clase "que más se acerque" a lo que usted necesita.*

Observe que leer un conjunto de declaraciones de clases derivadas puede ser confuso debido a que no se muestran los miembros heredados, sin embargo, están presentes en las clases derivadas. Puede existir un problema similar en la documentación de las clases derivadas.



**Observación de ingeniería de software 19.9**

*Una clase derivada contiene los atributos y el comportamiento de su clase base. Una clase derivada puede además contener atributos y comportamientos adicionales. Con la herencia, la clase base puede compilarse independientemente de la clase derivada. Solamente es necesario compilar los atributos y los comportamientos adicionales de la clase derivada para poder combinarlas con la clase base y formar una clase derivada.*



**Observación de ingeniería de software 19.10**

*Al modificar una clase base no es necesario modificar las clases derivadas, siempre y cuando las interfaces pública y protegida de la clase base permanezcan sin modificaciones. Sin embargo, podría ser necesario recompilar las clases derivadas.*

## 19.12 Composición *versus* herencia

Ya hemos explicado la relación *es un*, la cual es soportada por medio de la herencia pública. También ya explicamos la relación *tiene un* (y vimos ejemplos en los capítulos anteriores) en la cual, una clase puede tener otras clases como miembros; dichas relaciones crean nuevas clases por medio de la *composición* de clases existentes. Por ejemplo, dadas las clases **`Empleado`**, **`FechaNacimiento`** y **`NumeroTelefonico`**, es inapropiado decir que **`Empleado`** *es un* **`FechaNacimiento`** o que un **`Empleado`** *es un* **`NumeroTelefonico`**. Sin embargo, ciertamente es apropiado decir que cada **`Empleado`** *tiene una* **`FechaNacimiento`**, y que cada **`Empleado`** *tiene un* **`NumeroTelefonico`**.



**Observación de ingeniería de software 19.11**

*Las modificaciones de un programa a una clase que es miembro de otra clase no requiere que la clase que la contiene se modifique, siempre y cuando la interfaz pública de la clase miembro permanezca sin modificaciones. Sin embargo, observe que tal vez la clase compuesta necesite recompilarse.*

## 19.13 Relaciones *usa un* y *conoce un*

Tanto la herencia como la composición promueven la reutilización de software al crear nuevas clases que tienen mucho en común con las clases existentes. Existen otras formas de utilizar los servicios de las clases. Aunque un objeto **`persona`** no es un **`automóvil`** y una **`persona`** no contiene un **`automóvil`**, un objeto **`persona`** con certeza *usa un* **`automóvil`**. Una función utiliza un objeto al llamar a una función miembro no privada de ese objeto mediante el uso de un apuntador, una referencia o el mismo nombre del objeto.

Un objeto puede estar *conciente de* otro objeto. Con frecuencia, las redes de conocimiento tienen dichas relaciones. Un objeto puede contener un manipulador de apuntador o un manipulador de referencia hacia otro objeto para estar conciente de dicho objeto. En este caso se dice que un objeto tiene una relación *conoce un* objeto; en ocasiones a esto se le llama *asociación*.

## 19.14 Ejemplo práctico: `Punto`, `Circulo` y `Cilindro`

Consideremos ahora el ejercicio principal de este capítulo. Consideremos una jerarquía punto, círculo, cilindro. Primero desarrollamos y utilizamos la clase **`Punto`** (figura 19.8). Después presentamos un ejemplo en el cual derivamos la clase **`Circulo`** de la clase **`Punto`** (figura 19.8). Por último, presentamos un ejemplo en el cual derivamos la clase **`Cilindro`** a partir de la case **`Circulo`** (figura 19.10).

La figura 19.8 muestra la clase **`Punto`**. Las líneas 1 a 42 son el encabezado de la clase **`Punto`** y su archivo de implementación. Observe que los datos miembro de **`Punto`** son **`protected`**. Así, cuando se deriva la clase **`Circulo`** a partir de la case **`Punto`**, las funciones miembro de la clase **`Circulo`** serán capaces de hacer referencia directa a las coordenadas **`x`** y **`y`**, en lugar de utilizar funciones de acceso. Esto puede dar como resultado un mejor rendimiento.

```
// Figura 19.8: punto2.h
// Definición de la clase Punto
#ifndef PUNTO2_H
#define PUNTO2_H

#include <iostream>

using std::ostream;

class Punto {
   friend ostream &operator<<( ostream &, const Punto & );
public:
   Punto( int = 0, int = 0 );          // constructor predeterminado
   void establecePunto( int, int );    // establece las coordenadas
   int obtieneX() const { return x; }  // obtiene la coordenada x
   int obtieneY() const { return y; }  // obtiene la coordenada y
protected:        // accesible a las clases derivadas
   int x, y;      // coordenadas del punto
}; // fin de la clase punto

#endif
```

**Figura 19.8** Demostración de la clase **`Punto`**; **`punto2.h`**. (Parte 1 de 4.)

```
// Figura 19.8: punto2.cpp
// Funciones miembro para la clase Punto
#include “punto2.h”

// Constructor para la clase Punto
Punto::Punto( int a, int b ) { establecePunto( a, b ); }

// Establece las coordenadas x y y
void Punto::establecePunto( int a, int b )
{
   x = a;
   y = b;
} // fin de la función establecePunto

// Despliega Punto
ostream &operator<<( ostream &salida, const Punto &p )
{
   salida << ‘[‘ << p.x << “, “ << p.y << ‘]’;

   return salida;          // habilita la concatenación
} // fin de la función operator<<
```

**Figura 19.8** Demostración de la clase **`Punto`**; **`punto2.cpp`**. (Parte 2 de 4.)

```
// Figura 19.8: fig19_08.cpp
// Controlador para la clase Punto
#include <iostream>

```

**Figura 19.8** Demostración de la clase **`Punto`**; **`fig19_08.cpp`**. (Parte 3 de 4.)

```
47 using std::cout;
48 using std::endl;
49
50 #include “punto2.h”
51
52 int main()
53 {
54    Punto p( 72, 115 );   // crea la instancia del objeto p de Punto
55
56    // datos protegidos de Punto inaccesibles para main
57    cout << “la coordenada X es “ << p.obtieneX()
58         << “\nla coordenada Y es “ << p.obtieneY();
59
60    p.establecePunto( 10, 10 );
61    cout << “\n\nLa nueva ubicacion de p es “ << p << endl;
62
63    return 0;
64 } // fin de la función main
```

```
la coordenada X es 72
la coordenada Y es 115

La nueva ubicacion de p es [10, 10]
```

**Figura 19.8** Demostración de la clase **Punto**; **fig19_08.cpp**. (Parte 4 de 4.)

Las líneas 43 a 64 comprenden el programa controlador para la clase **Punto**. Observe que **main** debe utilizar las funciones de acceso **obtieneX** y **obtieneY** para leer los valores de los datos miembro protegidos **x** y **y**; recuerde que los datos miembro protegidos son accesibles solamente a los miembros y a las amigas de su clase, y a los miembros y las amigas de sus clases derivadas.

Nuestro siguiente ejemplo aparece en la figura 19.9. Aquí se reutiliza la definición de la clase **Punto** y la definición de las funciones miembro de la figura 19.8. Las líneas 1 a 62 muestran la definición de la clase **Circulo** y las definiciones de sus funciones miembro. Las líneas 63 a 90 son el programa controlador para la clase **Circulo**. Observe que la clase **Circulo** hereda desde la clase **Punto** mediante herencia pública. Esto significa que la interfaz pública de **Circulo** incluye las funciones miembro, así como las funciones miembro de **Circulo** **estableceRadio**, **obtieneRadio** y **area**.

```
1  // Figura 19.9: circulo2.h
2  // Definición de la clase Circulo
3  #ifndef CIRCULO2_H
4  #define CIRCULO2_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 #include “punto2.h”
11
12 class Circulo : public Punto {
13    friend ostream &operator<<( ostream &, const Circulo & );
14 public:
```

**Figura 19.9** Demostración de la clase **Circulo**; **circulo2.h**. (Parte 1 de 2.)

```
   // constructor predeterminado
   Circulo( double r = 0.0, int x = 0, int y = 0 );
   void estableceRadio( double ); // establece radio
   double obtieneRadio() const;   // devuelve el radio
   double area() const;           // calcula el área
protected:          // accesible a las clases derivadas
   double radio;   // radio del Circulo
}; // fin de la clase Circulo

#endif
```

**Figura 19.9** Demostración de la clase **`Circulo`**; **`circulo2.h`**. (Parte 2 de 2.)

```
// Figura 19.9: circulo2.cpp
// Definición de las funciones miembro para la clase Circulo
#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include “circulo2.h”

// El constructor para Circulo llama al constructor de Punto
// con el inicializador de miembros, e inicializa el radio
Circulo::Circulo( double r, int a, int b )
   : Punto( a, b )        // llama al constructor de la clase base
{ estableceRadio( r ); }

// Establece el radio
void Circulo::estableceRadio( double r )
   { radio = ( r >= 0 ? r : 0 ); }

// Obtiene el radio
double Circulo::obtieneRadio() const { return radio; }

// Calcula el área del Circulo
double Circulo::area() const
   { return 3.14159 * radio * radio; }

// Despliega un círculo con la forma:
// Centro = [x, y]; Radio = #.##
ostream &operator<<( ostream &salida, const Circulo &c )
{
   salida  << “Centro = “ << static_cast< Punto > ( c )
           << “; Radio = “
           << setiosflags( ios::fixed | ios::showpoint )
           << setprecision( 2 ) << c.radio;

   return salida;   //  permite las llamadas en cascada
} // fin de la función operator<<
```

**Figura 19.9** Demostración de la clase **`Circulo`**; **`circulo2.cpp`**.

```
// Figura 19.9: fig19_09.cpp
// Controlador para la clase Circulo
#include <iostream>

using std::cout;
using std::endl;

#include “punto2.h”
#include “circulo2.h”

int main()
{
   Circulo c( 2.5, 37, 43 );

   cout << “la coordenada X es “ << c.obtieneX()
        << “\nla coordenada Y es “ << c.obtieneY()
        << “\nEl radio es “ << c.obtieneRadio();

   c.obtieneRadio( 4.25 );
   c.obtienePunto( 2, 2 );
   cout << “\n\nLa nueva ubicacion y el radio de c es\n”
        << c << “\nArea “ << c.area() << ‘\n’;

   Punto &pRef = c;
   cout << “\nEl Circulo impreso como un Punto es: “ << pRef << endl;

   return 0;
} // fin de la función main
```

```
la coordenada X es 37
la coordenada Y es 43
El radio es 2.5

La nueva ubicacion y el radio de c es
Centro = [2, 2]; Radio = 4.25
Area 56.74

El Circulo impreso como un Punto es: [2, 2]
```

**Figura 19.9** Demostración de la clase **`Circulo`**; **`fig19_09.cpp`**.

Observe que la función del operador sobrecargado **`operator<<`**, como amiga de la clase **`Circulo`**, es capaz de mostrar la parte **`Punto`** de **`Circulo`** mediante la conversión de la referencia **`c`** de **`Circulo`** a **`Punto`**. Esto arroja como resultado una llamada a **`operator<<`** para **`Punto`** y despliega las coordenadas de **`x`** y **`y`** con el uso del formato apropiado para **`Punto`**.

El programa controlador crea la instancia de un objeto de la clase **`Circulo`** y utiliza funciones *obtener* para obtener la información acerca del objeto **`Circulo`**. De nuevo, **`main`** no es una función miembro ni una amiga de la clase **`Circulo`**, de modo que no puede hacer referencia directa a los datos protegidos de la clase **`Circulo`**. Después, el programa utiliza las funciones *establecer*, **`estableceRadio`** y **`establecePunto`** para reiniciar el radio y las coordenadas del centro del círculo. Por último, el controlador inicializa la referencia **`pRef`** de tipo “referencia a un objeto **`Punto`**” (**`Punto &`**) para el objeto **`c`** de **`Circulo`**. El controlador imprime entonces **`pRef`**, la cual, sin importar el hecho de que se inicializa con un objeto **`Circulo`**, “piensa” que es un objeto **`Punto`**, así que el objeto **`Circulo`** en realidad se imprime como un objeto **`Punto`**.

Nuestro último ejemplo aparece en la figura 19.10. Aquí reutilizamos las definiciones de la clase **`Punto`** y de la clase **`Circulo`**, así como las definiciones de sus funciones miembro correspondientes a las figuras 19.8 y 19.9. Las líneas 1 a 65 muestran la definición de la clase **`Cilindro`** y la definición de la función miembro **`Cilindro`**. Las líneas 66 a 109 son el programa controlador para la clase **`Cilindro`**. Observe que la clase **`Cilindro`** hereda de la clase **`Circulo`** mediante herencia pública. Esto significa que la interfaz pública para **`Cilindro`** incluye las funciones miembro de **`Circulo`** y las funciones miembro de **`Punto`**, así como las funciones miembro de **`Cilindro`** **`estableceAltura`**, **`obtieneAltura`**, **`area`** (redefinida de **`Circulo`**) y **`volumen`**. Observe que el constructor **`Cilindro`** es necesario para invocar al constructor de su clase base directa **`Circulo`**, pero no para su clase base indirecta **`Punto`**. Cada constructor de la clase derivada solamente es responsable de llamar a los constructores de la clase base inmediata a esa clase (o clases, en el caso de herencia múltiple). Además, observe que la función del operador sobrecargado **`operator<<`** de **`Cilindro`**, la cual es una amiga de la clase **`Cilindro`**, es capaz de desplegar la parte **`Circulo`** del **`Cilindro`** por medio de la conversión de la referencia **`c`** de **`Cilindro`** en un **`Circulo`**. Esto provoca una llamada a **`operator<<`** para **`Circulo`** y despliega las coordenadas **`x`** y **`y`**, y el **`radio`** por medio del formato adecuado para **`Circulo`**.

```
1  // Figura 19.10: cilindro2.h
2  // Definición de la clase Cilindro
3  #ifndef CILINDRO2_H
4  #define CILINDRO2_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 #include "circulo2.h"
11
12 class Cilindro : public Circulo {
13    friend ostream &operator<<( ostream &, const Cilindro & );
14
15 public:
16    // constructor predeterminado
17    Cilindro( double h = 0.0, double r = 0.0,
18              int x = 0, int y = 0 );
19
20    void estableceAltura( double ); // establece la altura
21    double obtieneAltura() const;   // devuelve la altura
22    double area() const;            // calcula y devuelve el área
23    double volumen() const;         // calcula y devuelve el volumen
24
25 protected:
26    double altura;                  // altura del cilindro
27 }; // fin de la clase cilindro
28
29 #endif
```

**Figura 19.10** Demostración de la clase **`Cilindro`**; **`cilindro2.h`**.

```
30 // Figura 19.10: cilindro2.cpp
31 // Definición de las funciones miembro y amigas
32 // para la clase Cilindro.
33 #include "cilindro2.h"
34
```

**Figura 19.10** Demostración de la clase **`Cilindro`**; **`cilindro2.cpp`**. (Parte 1 de 2.)

```
// El constructor de Cilindro llama al constructor de Circulo
Cilindro::Cilindro( double h, double r, int x, int y )
   : Circulo( r, x, y )   // llama al constructor de la clase base
{ estableceAltura( h ); }

// Establece la altura del Cilindro
void Cilindro::estableceAltura( double h )
   { altura = ( h >= 0 ? h : 0 ); }

// Obtiene la altura del Cilindro
double Cilindro::obtieneAltura() const { return altura; }

// Calcula el área del cilindro (es decir, la superficie)
double Cilindro::area() const
{
   return 2 * Circulo::area() +
          2 * 3.14159 * radio * altura;
} // fin de la función area

// Calcula el volumen del Cilindro
double Cilindro::volumen() const
   { return Circulo::area() * altura; }

// Despliega las dimensiones del Cilindro
ostream &operator<<( ostream &salida, const Cilindro &c )
{
   salida << static_cast< Circulo >( c )
          << “; Altura = “ << c.altura;

   return salida;   // permite llamadas en cascada
} // fin de la función operator<<
```

**Figura 19.10** Demostración de la clase **`Cilindro`**; **`cilindro2.cpp`**. (Parte 2 de 2.)

```
// Figura 19.10: fig19_10.cpp
// Controlador para la clase Cilindro
#include <iostream>

using std::cout;
using std::endl;

#include “punto2.h”
#include “circulo2.h”
#include “cilindro2.h”

int main()
{
   // crea el objeto Cilindro
   Cilindro cilin( 5.7, 2.5, 12, 23 );

   // utiliza funciones obtener para desplegar el Cilindro
   cout << “La coordenada X es “ << cilin.obtieneX()
        << “\nLa coordenada Y es “ << cilin.obtieneY()
        << “\nEl radio es “ << cilin.obtieneRadio()
        << “\nLa altura es “ << cilin.obtieneAltura() << “\n\n”;
```

**Figura 19.10** Demostración de la clase **`Cilindro`**; **`fig19_10.cpp`**. (Parte 1 de 2.)

```

    // utiliza funciones establecer para modificar los atributos del Cilindro
    cilin.estableceAltura( 10 );
    cilin.estableceRadio( 4.25 );
    cilin.establecePunto( 2, 2 );
    cout << “La nueva ubicacion, radio, y altura de cilin es:\n”
         << cilin << ‘\n’;

    cout << “El area de cilin es:\n”
         << cilin.area() << ‘\n’;

    // despliega el Cilindro como un Punto
    Punto &pRef = cilin;   // pRef “piensa” que es un punto
    cout << “\nEl Cilindro impreso como un punto es: “
         << pRef << “\n\n”;

    // despliega el Cilindro como un Circulo
    Circulo &refCirculo = cilin;  // refCirculo piensa que es un Circulo
    cout << “El Cilindro impreso como un Circulo es:\n” << refCirculo
         << “\nArea: “ << refCirculo.area() << endl;

    return 0;
} // fin de la función main
```

```
La coordenada X es 12
La coordenada Y es 23
El radio es 2.5
La altura es 5.7

La nueva ubicacion, radio, y altura de cilin es:
Centro = [2, 2]; Radio = 4.25; Altura = 10.00
El area de cilin es:
380.53

El Cilindro impreso como un punto es: [2, 2]

El Cilindro impreso como un Circulo es:
Centro = [2, 2]; Radio = 4.25
Area: 56.74
```

**Figura 19.10** Demostración de la clase **`Cilindro`**; **`fig19_10.cpp`**. (Parte 2 de 2.)

El programa controlador crea una instancia del objeto de la clase **`Cilindro`** y después utiliza funciones *obtener* para obtener la información acerca del objeto **`Cilindro`**. De nuevo, **`main`** no es ni una función miembro ni una amiga de la clase **`Cilindro`**, de modo que no puede hacer referencia directa a los datos protegidos de la clase **`Cilindro`**. El programa controlador utiliza las funciones *establecer* **`estableceAltura`**, **`estableceRadio`** y **`establecePunto`** para restablecer la altura, el radio y las coordenadas del cilindro. Por último, el controlador inicializa la variable de referencia **`pRef`**, de tipo “referencia a un objeto **`Punto`**” (**`Punto&`**), hacia el objeto **`cilin`** de **`Cilindro`**. Posteriormente imprime **`pRef`**, la cual, sin importar el hecho de que se inicializa con el objeto **`Cilindro`**, “piensa” que es un objeto **`Punto`**, de modo que el objeto **`Cilindro`** se imprime en realidad como un objeto **`Punto`**. El controlador después inicializa la referencia **`refCirculo`** de tipo “referencia al objeto **`Circulo`**” (**`Circulo &`**) hacia el objeto **`cilin`** de **`Cilindro`**. El programa controlador posteriormente imprime **`refCirculo`**, la cual, a pesar del hecho de que se inicializa con un objeto **`Cilindro`**, “piensa” que es un objeto **`Circulo`**, por lo que el objeto **`Cilindro`** en realidad se imprime como un objeto **`Circulo`**. También despliega el área del círculo.

El ejemplo demuestra claramente la herencia pública y la definición de referencias a datos miembro **`protected`**. Ahora, usted debe sentirse seguro de los principios de la herencia. En el siguiente capítulo, mostraremos cómo programar con el uso de jerarquías de herencia de una manera general mediante el uso del polimorfismo. La abstracción de datos, la herencia y el polimorfismo son la base de la programación orientada a objetos.

## *RESUMEN*

- Una de las claves del poder de la programación orientada a objetos es lograr la reutilización de software a través de la herencia.
- El programador puede definir que la nueva clase herede los datos y las funciones miembro de una clase base previamente definida. En este caso, a la nueva clase se le conoce como una clase derivada.
- Con la herencia simple, una clase hereda solamente de una clase base. Con herencia múltiple, una clase derivada hereda de varias (posiblemente no relacionadas) clases base.
- Por lo general, una clase derivada contiene datos y funciones miembro propias, de modo que las clases derivadas tienen una definición más grande que su clase base. Una clase derivada es más específica que su clase base y, por lo general, representa a menos objetos.
- Una clase derivada no tiene acceso a los miembros privados de su clase base; permitir esto violaría el encapsulamiento de la case base. Sin embargo, una clase derivada puede acceder a los miembros públicos y privados de su clase base.
- El constructor de una clase derivada siempre llama al constructor de su clase base para crear e inicializar las clases derivadas miembro de la clase base.
- Los destructores se invocan en orden inverso a las llamadas de los destructores, así que el destructor de una clase derivada se llama antes que el destructor de su clase base.
- La herencia permite la reutilización de software, la cual ahorra tiempo de desarrollo y fortalece el uso de software previamente probado y de alta calidad.
- La herencia se puede llevar a cabo a partir de bibliotecas de clases existentes.
- Algún día la mayor parte del software se construirá a partir de componentes estándares reutilizables, tal como se construye la mayoría del hardware hoy en día.
- El implementador de una clase derivada no necesita tener acceso al código fuente de la clase base, pero sí necesita la interfaz de su clase base y el código objeto de su clase base.
- Un objeto de una clase derivada puede tratarse como un objeto de su clase base pública correspondiente. Sin embargo, lo contrario no es cierto.
- Una clase base existe en una relación jerárquica con sus clases derivadas.
- Una clase puede existir por sí misma. Cuando se utiliza la clase con el mecanismo de la herencia, se puede convertir en una clase base que proporciona atributos y comportamientos a otras clases, o en una clase derivada que hereda dichos atributos y comportamientos.
- Una jerarquía de herencia puede ser tan profunda como lo permitan las limitaciones de un sistema en particular.
- Las jerarquías son herramientas útiles para comprender y manipular la complejidad del software. Con software cada vez más complejo, C++ proporciona mecanismos para soportar estructuras jerárquicas a través de la herencia y el polimorfismo.
- Se puede utilizar una conversión explícita para convertir un apuntador de una clase base en un apuntador de una clase derivada. Dicho apuntador no se debe desreferenciar, a menos que apunte a un objeto del tipo de la clase derivada.
- El acceso **`protected`** (protegido) sirve como nivel de protección intermedio entre el acceso **`public`** (público) y el acceso **`private`** (privado). Se puede acceder a los miembros protegidos de una clase base mediante miembros y amigas de la clase base, y meidante miembros y amigas de las clases derivadas; ninguna otra función puede acceder a los miembros protegidos de una clase base.
- Los miembros protegidos se utilizan para extender los privilegios a las clases derivadas, mientras restringe dichos privilegios a las funciones que no son de la clase, o amigas de la clase.
- Cuando se deriva una clase de una clase base, la clase base puede declararse como pública, protegida o privada.
- Cuando se deriva una clase a partir de una clase base pública, los miembros públicos de la clase base se hacen miembros públicos de la clase derivada, y los miembros protegidos de la clase base se vuelven miembros protegidos de la clase derivada.

- Cuando se deriva una clase a partir de una clase base protegida, los miembros públicos y protegidos de la clase base se hacen miembros protegidos de la clase derivada.
- Cuando se deriva una clase a partir de una clase base privada, los miembros públicos y protegidos de la clase base se hacen miembros privados de la clase derivada.
- Una clase base puede ser una clase base directa de una clase derivada, o una clase base indirecta de una clase derivada. Una clase base directa se lista explícitamente en donde se declara la clase derivada. Una clase base indirecta no se lista de manera explícita; en vez de eso, se hereda de varios niveles superiores del árbol de jerarquía de la clase.
- Cuando un miembro de clase base no es apropiado para una clase derivada, simplemente podemos redefinir a dicho miembro en la clase derivada.
- Es importante distinguir entre una relación *es un* y una relación *tiene un*. En una relación *tiene un*, el objeto de una clase tiene como miembro un objeto de otra clase. En una relación *es un*, un objeto de la clase derivada puede tratarse también como un objeto del tipo de la clase base. *Es un* es herencia, mientras que *tiene un* es composición.
- Un objeto de una clase derivada puede asignarse a un objeto de una clase base. Este tipo de asignación tiene sentido debido a que la clase derivada tiene miembros que corresponden, cada uno, a los miembros de la clase base.
- Un apuntador a un objeto de una clase derivada puede convertirse implícitamente en un apuntador a un objeto de la clase base.
- Es posible convertir un apuntador de una clase base en un apuntador de una clase derivada por medio de una conversión explícita. El destino debe ser un objeto de la clase derivada.
- Una clase base especifica similitudes. Todas las clases derivadas desde una clase base heredan las capacidades de dicha clase base. En el proceso del diseño orientado a objetos, el diseñador busca las similitudes y las aprovecha para formar clases base apropiadas. Las clases derivadas entonces se personalizan más allá de las capacidades heredadas de su clase base.
- Leer un conjunto de declaraciones de clases derivadas puede ser confuso debido a que no todos los miembros de la clase derivada están presentes en estas declaraciones. En especial, los miembros heredados no se listan en las declaraciones de clases derivadas, pero estos miembros en realidad están presentes en las clases derivadas.
- Las relaciones *tiene un* son ejemplos de la creación de nuevas clases por medio de la composición de clases existentes.
- Las relaciones *conoce un* son ejemplos de objetos que contienen apuntadores o referencias a otros objetos, de modo que pueden estar concientes de dichos objetos.
- A los constructores de objetos miembro se les llama en el orden en el que se declaran los objetos. En la herencia, los constructores de las clases base se llaman en el orden en el que se especifica la herencia, y antes del constructor de la clase derivada.
- Para un objeto de la clase derivada, primero se llama al constructor de la clase base, y luego se llama al constructor de la clase derivada (el cual puede llamar a los constructores de los objetos miembro).
- Cuando se destruye un objeto de una clase derivada, se llama a los destructores en el orden inverso a los constructores, primero se llama al destructor de la clase derivada, y luego se llama al destructor de la clase base.
- Una clase puede derivarse de más de una clase base; tal derivación se denomina herencia múltiple.
- Indique la herencia múltiple colocando una lista separada por comas de las clases base después del indicador de herencia (`:`).
- El constructor de la clase derivada llama a los constructores de las clases base mediante la sintaxis de inicialización de miembros. Los constructores de la clase base se llaman en el orden en el que se declaran las clases base durante la herencia.

## TERMINOLOGÍA

abstracción
amiga de una clase base
amiga de una clase derivada
apuntador a un objeto de clase base
apuntador a un objeto de clase derivada
apuntador de clase base
apuntador de clase derivada
asociación
biblioteca de clases
clase base
clase base directa
clase base indirecta
clase base privada
clase base protegida
clase base pública
clase derivada
clase miembro
cliente de una clase
componentes estándares de software
composición
constructor de clase base
constructor de clase derivada
constructor predeterminado de clase base
control de acceso a miembros

conversión hacia abajo de un apuntador
conversión hacia arriba de un apuntador
destructor de clase base
destructor de clase derivada
herencia
herencia múltiple
herencia privada
herencia protegida
herencia pública
herencia simple
inicializador de clase base
jerarquía de la clases
miembro protegido de una clase
objeto miembro
palabra reservada **`protected`**
programación orientada a objetos (POO)
redefinición de una función
redefinir una función miembro de una clase base
relación *conoce un*
relación *es un*
relación jerárquica
relación *tiene un*
relación *usa un*
reutilización de software
software personalizado
subclase
superclase

## ERRORES COMUNES DE PROGRAMACIÓN

**19.1** Tratar un objeto de la clase base como un objeto de la clase derivada puede provocar errores.

**19.2** Convertir explícitamente un apuntador de una clase base que apunta a un objeto de la clase base en un apuntador de clase derivada, y después hacer referencia a los miembros de la clase derivada que no existen en dicho objeto, puede provocar errores lógicos en tiempo de ejecución.

**19.3** Cuando en una clase derivada se redefine una función miembro de la clase base, es común hacer que la versión de la clase derivada llame a la versión de la clase base y hacer algo de trabajo adicional. No utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base provoca una recursividad infinita, ya que la función miembro de la clase derivada en realidad se llama a sí misma. Esto provocará que en algún momento se agote la memoria del sistema; un error fatal en tiempo de ejecución.

**19.4** Asignar un objeto de clase derivada a un objeto de su clase base correspondiente, y luego intentar hacer referencia a miembros exclusivos de la clase derivada en el nuevo objeto de la clase base, es un error de sintaxis.

**19.5** Convertir un apuntador de clase base en un apuntador de clase derivada puede provocar errores si dicho apuntador se utiliza para hacer referencia a un objeto de la clase base que no tiene los miembros requeridos en la clase derivada.

## TIPS DE RENDIMIENTO

**19.1** Cuando el rendimiento es un asunto de mayor importancia, es posible que los programadores deseen ver el código fuente de las clases de las que heredan, de modo que puedan poner a punto el código para que cumpla con sus requerimientos de rendimiento.

**19.2** Si las clases producidas a través de la herencia son más grandes de lo necesario, los recursos de memoria y programación pueden desperdiciarse. Herede de la clase “que más se acerque” a lo que usted necesita.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**19.1** En general, declare los datos miembro de una clase como **`private`** y utilice **`protected`** solamente como “último recurso”, cuando los sistemas necesiten cumplir ciertos requerimientos de rendimiento.

**19.2** Una clase derivada no puede acceder directamente a los miembros privados de su clase base.

**19.3** Suponga que creamos un objeto de una clase derivada, en donde tanto la clase base como la clase derivada contienen objetos de otras clases. Cuando se crea un objeto de dicha clase derivada, primero se ejecutan los constructores de los objetos miembros de la clase base, luego se ejecutan los constructores de la clase base, después se ejecutan los constructores de los objetos miembro de la clase derivada, y por último se llama a sus constructores correspondientes.

**19.4** El orden en el cual se construyen los objetos miembro es el orden en el que se declaran dichos objetos dentro de la definición de la clase. El orden en el cual los inicializadores de miembros se listan no afecta el orden de construcción.

**19.5** En la herencia, los constructores de la clase base se llaman en el orden en el que se especifica la herencia en la definición de la clase derivada. El orden en el cual se especifican los constructores de la clase base en la lista de inicialización de miembros de la clase derivada, no afecta el orden de la construcción.

**19.6** En teoría, los usuarios no necesitan ver el código fuente de las clases de las cuales heredan. En la práctica, la gente que vende licencias de las clases nos ha dicho que con frecuencia los clientes requieren el código fuente. Al pa-

recer, los programadores son reticentes a incorporar código dentro de sus programas cuando este código fue escrito por otras personas.

**19.7** La creación de una clase derivada no afecta el código fuente o el código objeto de su clase base; la integridad de la clase base se preserva mediante la herencia.

**19.8** En un sistema orientado a objetos, con frecuencia las clases están íntimamente relacionadas. "Descubra" los atributos y los comportamientos comunes y colóquelos en una clase. Después utilice la herencia para formar clases derivadas.

**19.9** Una clase derivada contiene los atributos y el comportamiento de su clase base. Una clase derivada puede además contener atributos y comportamientos adicionales. Con la herencia, la clase base puede compilarse independientemente de la clase derivada. Solamente es necesario compilar los atributos y los comportamientos adicionales de la clase derivada para poder combinarlas con la clase base y formar una clase derivada.

**19.10** Al modificar una clase base no es necesario modificar las clases derivadas, siempre y cuando las interfaces pública y protegida de la clase base permanezcan sin modificaciones. Sin embargo, podría ser necesario recompilar las clases derivadas.

**19.11** Las modificaciones de un programa a una clase que es miembro de otra clase no requiere que la clase que la contiene se modifique, siempre y cuando la interfaz pública de la clase miembro permanezca sin modificaciones. Sin embargo, observe que tal vez la clase compuesta necesite recompilarse.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**19.1** Complete los espacios en blanco:

a) Si la clase **`Alfa`** hereda de la clase **`Beta`**, a la clase **`Alfa`** se le llama ____________, y a la clase **`Beta`** se le llama ____________.

b) C++ proporciona la ____________, la cual permite a una clase derivada heredar de muchas clases base, incluso si estas clases base no están relacionadas entre sí.

c) La herencia permite la ____________, la cual ahorra tiempo de desarrollo y promueve el uso de software de alta calidad ya creado.

d) Un objeto de clase ____________ puede tratarse como un objeto de su clase ____________ correspondiente.

e) Para convertir un apuntador de una clase base en un apuntador de una clase derivada, se debe utilizar una ____________ debido a que el compilador considera que ésta es una operación peligrosa.

f) Los tres especificadores de acceso a miembros son ____________, ____________ y ____________.

g) Cuando se derivan clases a partir de una clase base con herencia múltiple, los miembros públicos de la clase base se hacen miembros ____________ de la clase derivada, y los miembros protegidos de la clase base se hacen miembros ____________ de la clase derivada.

h) Cuando se derivan clases a partir de una clase base con herencia protegida, los miembros públicos de la clase base se hacen miembros ____________ de la clase derivada, y los miembros protegidos de la clase base se hacen miembros ____________ de la clase derivada.

i) Una relación *tiene un* entre clases representa la ____________ y una relación *es un* entre clases representa la ____________.

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**19.1** a) Derivada, base. b) Herencia múltiple. c) Reutilización de software. d) Derivada, base. e) Conversión. f) Pública, protegida, privada. g) Públicos, protegidos. h) Protegidos, protegidos. i) Composición, herencia.

## *EJERCICIOS*

**19.2** Considere la clase **`Bicicleta`**. Dado su conocimiento acerca de algunos componentes de las bicicletas, muestre una jerarquía de clases en la que la clase **`Bicicleta`** herede desde otras clases, las cuales, a su vez, hereden desde otras clases. Explique la generación de instancias de varios objetos de la clase **`Bicicleta`**. Explique la herencia de la clase **`Bicicleta`** para otras clases derivadas muy relacionadas.

**19.3** Defina brevemente cada uno de los siguientes términos: herencia, herencia múltiple, clase base y clase derivada.

**19.4** Explique por qué el compilador considera peligrosa a la conversión de un apuntador de una clase base a un apuntador de una clase derivada.

**19.5** (Verdadero/Falso.) Con frecuencia, a una clase derivada se le llama subclase debido a que representa un subconjunto de su clase base (es decir, una clase derivada es por lo general más pequeña que su clase base).

**19.6** (Verdadero/Falso.) Un objeto de una clase derivada es también un objeto de la clase base de dicha clase derivada.

**19.7** Algunos programadores prefieren no utilizar el acceso protegido debido a que viola el encapsulamiento de la clase base. Explique los méritos del acceso protegido contra la insistencia de utilizar el acceso privado en las clases base.

**19.8** Muchos programas escritos con herencia pueden resolverse mediante la composición, y viceversa. Explique los méritos de estos métodos en el contexto de la jerarquía de las clases **`Punto`**, **`Circulo`**, **`Cilindro`** de este capítulo. Rescriba el programa de la figura 19.10 (y las clases soportadas) para utilizar la composición en lugar de la herencia. Después de hacer esto, insista en los méritos de los dos métodos tanto para el problema del **`Punto`**, **`Circulo`**, **`Cilindro`**, y en general de los programas orientados a objetos.

**19.9** En este capítulo, dijimos que "cuando una clase base miembro no es apropiada para una clase derivada, ese miembro puede redefinirse en la clase derivada, con una implementación apropiada". Si se hace esto, ¿se mantiene la relación de clase derivada *es un* objeto de la clase base? Explique su respuesta.

**19.10** Estudie la jerarquía de herencia de la figura 19.2. Para cada clase, indique algunos atributos y comportamientos comunes, consistentes con la jerarquía. Agregue algunas otras clases (**`EstudianteTitulado`**, **`EstudianteGraduado`**, **`DePrimerAnio`**, **`DeSegundoAnio`**, **`DeTerceranio`**, **`DeCuartoAnio`**, etcétera) para enriquecer la jerarquía.

**19.11** Escriba una jerarquía de herencia para la clase **`Cuadrilatero`**, **`Trapezoide`**, **`Paralelogramo`**, **`Rectangulo`** y **`Cuadrado`**. Utilice **`Cuadrilatero`** como la clase base de la jerarquía. Haga la jerarquía tan profunda (es decir, que tenga tantos niveles) como sea posible. Los datos privados de **`Cuadrilatero`** deben ser los pares de coordenadas ($x$, $y$) para las cuatro esquinas de **`Cuadrilatero`**. Escriba un programa controlador que genere una instancia y que despliegue los objetos de cada una de esas clases.

**19.12** Escriba todas las figuras que se le ocurran, tanto de dos como de tres dimensiones, y diseñe dichas figuras dentro de la jerarquía de figuras. Su jerarquía debe tener una clase base **`Figura`** de la que se deriven la clase **`FiguraBidimensional`** y la clase **`FiguraTridimensional`**. Una vez que desarrolle la jerarquía, defina cada una de las clases en la jerarquía. Esta jerarquía la utilizaremos en los ejercicios del capítulo 20 para procesar todas las figuras como objetos de la clase base **`Figura`**. Ésta es una técnica llamada polimorfismo.

# 20

# Funciones virtuales y polimorfismo en C++

## Objetivos

- Comprender el concepto de polimorfismo.
- Comprender cómo declarar y utilizar las funciones **`virtuales`** para efectos de polimorfismo.
- Comprender la diferencia entre clases abstractas y concretas.
- Aprender cómo declarar funciones **`virtuales`** puras para crear clases abstractas.
- Apreciar cómo el polimorfismo hace que los sistemas se puedan ampliar y sean más fáciles de mantener.
- Comprender cómo es que C++ implementa las funciones **`virtuales`** y cómo realiza la vinculación dinámica “tras bambalinas”.




*Un anillo para gobernarlos a todos, un anillo para encontrarlos,*
*Un anillo para traerlos, y en la oscuridad atarlos.*
John Ronald Reuel Tolkien

*Con frecuencia, el silencio de la inocencia pura*
*Persuade cuando el habla falla.*
William Shakespeare

*Las proposiciones generales no deciden casos concretos.*
Oliver Wendell Holmes

*Un filósofo de imponente estatura no piensa en un vacío.*
*Incluso sus ideas más abstractas son, hasta cierto punto,*
*condicionadas por lo que se sabe, o no se sabe, en la época*
*en la que vive.*
Alfred North Whitehead

## Plan general



## 20.1 Introducción

Con funciones virtuales y polimorfismo, es posible diseñar e implementar sistemas que sean más fáciles de ampliar. Los programas pueden escribirse para procesar genéricamente (como objetos de clase base) objetos de todas las clases existentes en una jerarquía. Las clases que no existen durante el desarrollo de un programa pueden añadirse con muy poca o ninguna modificación a la parte genérica del programa, mientras esas clases sean parte de la jerarquía que se está procesando genéricamente. Las únicas partes de un programa que necesitarán modificación son aquellas que requieren un conocimiento directo de la clase en particular que se agrega a la jerarquía.

## 20.2 Tipos de campos e instrucciones `switch`

Una manera de manejar objetos de diferentes tipos es por medio de una instrucción **`switch`** que efectúe acciones adecuadas sobre cada objeto, basándose en el tipo de ese objeto. Por ejemplo, en una jerarquía de formas, en la que cada forma especifica su tipo como un dato miembro, una estructura **`switch`** podría determinar a qué función **`imprimir`** llamar, basándose en el tipo del objeto en particular.

Existen muchos problemas con el uso de la lógica de **`switch`**. El programador podría olvidar realizar una evaluación de tipo, cuando uno está garantizado. El programador podría olvidar evaluar todos los casos posibles de un **`switch`**. Si se modifica un sistema basado en **`switch`**, agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en todas las instrucciones **`switch`** existentes. Toda adición o eliminación de una clase para manejar nuevos tipos, requiere que se modifique toda instrucción **`switch`** en el sistema; dar seguimiento a esto puede consumir demasiado tiempo y es propenso a errores.

Como veremos, las funciones virtuales y la programación polimórfica puede eliminar la necesidad de la lógica de **`switch`**. El programador puede utilizar el mecanismo de la función **`virtual`** para realizar el equivalente lógico, lo que evitaría los tipos de errores generalmente asociados con la lógica de **`switch`**.



**Observación de ingeniería de software 20.1**

*Una consecuencia interesante de utilizar funciones virtuales y el polimorfismo es que los programas adquieren una apariencia simplificada. Éstos contienen menos divisiones lógicas, a favor de código secuencial más sencillo. Esto facilita la evaluación, la depuración y el mantenimiento de programas, así como la eliminación de errores.*

## 20.3 Funciones `virtuales`

Suponga que un conjunto de clases de figuras tales como **`Circulo`**, **`Triangulo`**, **`Rectangulo`**, **`Cuadrado`**, etcétera, se derivan de la clase base **`Figura`**. En la programación orientada a objetos, cada una de estas

clases podría dotarse con la habilidad de dibujarse a sí mismas. Aunque cada clase tiene su propia función dibujar, la función **`dibujar`** para cada figura es muy diferente. Cuando se dibuja una figura, cualquiera que ésta sea, sería bueno poder tratar a todas las figuras de manera genérica como objetos de la clase base **`Figura`**. Entonces, para dibujar cualquier figura, podríamos simplemente llamar a la función **`dibujar`** de la clase base **`Figura`**, y dejar que el programa determine *dinámicamente* (es decir, en tiempo de ejecución) qué clase derivada de la función **`dibujar`** debe utilizar.

Para permitir esta clase de comportamiento, declaramos a **`dibujar`** en la clase base como una *función virtual* y la *pasamos por alto* en cada una de las clases derivadas para dibujar la figura apropiada. Una función **`virtual`** se declara precediendo al prototipo de la función con la palabra reservada **`virtual`** en la clase base. Por ejemplo,

```
virtual void dibujar() const;
```

puede aparecer en la clase base **`Figura`**. El prototipo anterior declara que la función **`dibujar`** es una función constante que no toma argumentos, que devuelve nada y que es una función **`virtual`**.



**Observación de ingeniería de software 20.2**

*Una vez que una función se declara como virtual, ésta permanece así en todos los niveles inferiores de la jerarquía de herencia a partir de ese punto, incluso si no se le declara como virtual cuando una clase la sustituye.*

**Buena práctica de programación 20.1**

*Aun cuando ciertas funciones son implícitamente virtuales, debido a una declaración hecha en un nivel superior de la jerarquía de la clase, declare explícitamente estas funciones como* **`virtual`** *en cada nivel de la jerarquía para promover la claridad del programa.*

**Observación de ingeniería de software 20.3**

*Cuando una clase derivada elige no definir una función* **`virtual`***, la clase derivada simplemente hereda la definición de la función* **`virtual`** *de la clase base inmediata.*

Si la función **`dibujar`** de la clase base se declaró como **`virtual`**, y si después utilizamos un apuntador de la clase base o una referencia para apuntar al objeto de la clase base derivada, e invocamos a la función **`dibujar`** por medio de este apuntador (por ejemplo, **`ptrFigura->dibujar()`**) o referencia, el programa elegirá dinámicamente (es decir, en tiempo de ejecución) a la función dibujar de la clase derivada correcta, basándose en el tipo de objeto; no en el tipo del apuntador o referencia. En el ejemplo práctico de la sección 20.8, ilustraremos tal *vinculación dinámica*.

Cuando se llama a una función **`virtual`**, haciendo referencia a un objeto específico por su nombre y utilizando el operador punto de selección de miembros (por ejemplo, **`objetoCuadrado.dibujar()`**), la referencia se resuelve en tiempo de compilación (a esto se le llama *vinculación estática*), y la función **`virtual`** que se invoca es la definida (o heredada) por la clase de ese objeto en particular.

## 20.4 Clases base abstractas y clases concretas

Cuando pensamos en una clase como un tipo, asumimos que se generarán instancias de los objetos de ese tipo. Sin embargo, existen casos en los que es útil definir clases para las que el programador nunca intenta instanciar objeto alguno. Dichas clases se conocen como *clases abstractas*. Éstas se utilizan como clases base en situaciones de herencia, por lo que normalmente nos referiremos a ellas como *clases base abstractas*. Ningún objeto de una clase base abstracta puede instanciarse.

El único propósito de una clase abstracta es el de proporcionar una clase base apropiada, a partir de la cual, las clases pueden heredar la interfaz y/o la implementación. Las clases cuyos objetos pueden instanciarse se conocen como *clases concretas*.

Podríamos tener una clase base abstracta **`FiguraBidimensional`**, y derivar clases concretas como **`Cuadrado`**, **`Circulo`**, **`Triangulo`**, etcétera. También podríamos tener una clase base abstracta **`FiguraTridimensional`**, y derivar clases concretas como **`Cubo`**, **`Esfera`**, **`Cilindro`**, etcétera. Las clases base abstractas son demasiado genéricas como para definir objetos reales; necesitamos ser más específicos antes de pensar en instanciar objetos. Esto es lo que hacen las clases concretas; éstas proporcionan las especificaciones que hacen razonable instanciar objetos.

Una clase se hace abstracta, declarando una o más de sus funciones virtuales para que sean "puras". Una *función virtual pura* es aquella que tiene un *inicializador* =0 en su declaración, como en el caso de

```
virtual double utilidades() const = 0; // función virtual pura
```

**Observación de ingeniería de software 20.4**

*Si una clase se deriva de una clase con una función* **`virtual`** *pura, y si no se proporciona una definición para dicha función en la clase derivada, entonces esa función virtual permanece pura en la clase derivada. En consecuencia, la clase derivada también es una clase abstracta.*



**Error común de programación 20.1**

*Intentar crear una instancia de un objeto correspondiente a una clase abstracta (es decir, una clase que contiene una o más funciones virtuales), es un error de sintaxis.*

Una jerarquía no necesita contener clases abstractas, sin embargo, como veremos, muchos buenos sistemas orientados a objetos tienen jerarquías de clases encabezadas por una clase base abstracta. En algunos casos, las clases abstractas constituyen la cima de los niveles de la jerarquía. Un buen ejemplo de esto es una jerarquía de figuras. La jerarquía podría estar encabezada por la clase base abstracta **`Figura`**. En el siguiente nivel, podemos tener dos clases base abstractas adicionales; a saber, **`FiguraBidimensional`** y **`FiguraTridimensional`**. El siguiente nivel hacia abajo comenzaría con la definición de clases concretas para las figuras bidimensionales, como círculos y cuadrados, y de clases concretas para figuras tridimensionales, como esferas y cubos.

## 20.5 Polimorfismo

C++ permite el *polimorfismo*; la habilidad de los objetos de diferentes clases relacionadas por la herencia de responder de manera diferente al mismo mensaje (es decir, a una llamada de una función miembro). El mismo mensaje enviado a muchos tipos diferentes de objetos toma "muchas formas"; de aquí el término polimorfismo. Por ejemplo, si la clase **`Rectangulo`** se deriva de **`Cuadrilateros`**, entonces un objeto **`Rectangulo`** *es* una versión más específica de un objeto **`Cuadrilateros`**. Una operación (por ejemplo, el cálculo del perímetro) que puede realizarse sobre un objeto **`Cuadrilateros`** también puede realizarse sobre un objeto **`Rectangulo`**.

El polimorfismo se implementa a través de funciones virtuales. Cuando se hace una solicitud por medio de un apuntador (o referencia) de clase base para utilizar una función **`virtual`**, C++ elige la función correcta a ignorar de la clase derivada asociada con el objeto.

Algunas veces se define una función miembro no virtual en una clase base y se ignora en una clase derivada. Si se llama a dicha función miembro a través de un apuntador de clase base hacia el objeto de clase derivada, se utiliza la versión de la clase base. Si se llama a la función miembro a través de un apuntador de clase derivada, se utiliza la versión de la clase derivada. Éste es un comportamiento no polimórfico.

Considere el siguiente ejemplo, el cual utiliza la clase base **`Empleado`** y la clase derivada **`EmpPorHoras`** de la figura 19.5:

```
Empleado e, *ptrE = &e;
EmpPorHoras h, *ptrH = &h;
ptrE->imprime ();       // llama a la clase base de la función imprime
ptrH-> imprime ();      // llama a la clase derivada de la función imprime
ptrE = &h;              // conversión implícita permisible
ptrE-> imprime ();      // aún llama a la clase base de imprime
```

Nuestra clase base **`Empleado`** y la clase derivada **`EmpPorHoras`** tienen sus propias funciones imprime definidas. Las funciones no se declararon como **`virtual`**, y tienen la misma firma, por lo que llamar a la función imprime a través de un apuntador **`Empleado`** da como resultado la llamada a **`Empleado:: imprime()`** (independientemente de si el apuntador **`Empleado`** apunta hacia un objeto de la clase base **`Empleado`**, o a un objeto de la clase derivada **`EmpPorHoras`**), y llamar a la función imprime a través de un apuntador **`EmpPorHoras`** da como resultado la llamada a la función **`EmpPorHoras`**:: imprime (). La clase base de la función imprime también está disponible para la clase derivada, pero por ejemplo, para llamar a la clase base imprime

para un objeto de la clase derivada a través de un apuntador a un objeto de la clase derivada, la función debe llamarse explícitamente de la siguiente manera:

```
ptrH->Empleado:: imprime ();  //llama a la clase base de la función imprime
```

Esto especifica que la clase base imprime debe llamarse explícitamente.

Por medio del uso de las funciones virtuales y del polimorfismo, una llamada a una función miembro ocasiona diferentes acciones de acuerdo con el tipo de objeto que recibe la llamada (veremos que se necesita un poco de sobrecarga en tiempo de ejecución). Esto da al programador una capacidad de expresión enorme. En las siguientes secciones, veremos ejemplos del poder del polimorfismo y de las funciones virtuales.



**Observación de ingeniería de software 20.5**

*Con las funciones virtuales y el polimorfismo, el programador puede manejar generalidades y dejar que el ambiente en tiempo de ejecución se ocupe de las particularidades. El programador puede manejar una amplia variedad de objetos para que se comporten de manera apropiada, sin siquiera tener que conocer los tipos de esos objetos.*



**Observación de ingeniería de software 20.6**

*El polimorfismo promueve la extensibilidad: el software escrito para invocar un comportamiento polimórfico se escribe de manera independiente de los tipos de los objetos a los que se envían los mensajes. Entonces, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en un sistema, sin tener que modificar el sistema base. Con excepción del código cliente que genera instancias de nuevos objetos, los programas no necesitan recompilarse.*



**Observación de ingeniería de software 20.7**

*Una clase abstracta define una interfaz para los diferentes miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras que se definirán en las clases derivadas. Todas las funciones de la jerarquía pueden utilizar esta misma interfaz, a través del polimorfismo.*

Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar apuntadores y referencias hacia clases base abstractas. Tales apuntadores y referencias pueden entonces utilizarse para permitir manipulaciones polimórficas de los objetos de clases derivadas, cuando dichos objetos son instanciados a partir de clases concretas.

Consideremos aplicaciones del polimorfismo y de las funciones virtuales. Un administrador de pantalla necesita desplegar muchos objetos de diferentes clases, incluso nuevos tipos de objetos que se agregarán al sistema, incluso después de que se haya escrito el administrador de pantalla. El sistema puede necesitar desplegar varias figuras (es decir, la clase base es **`Figura`**) como cuadrados, círculos, triángulos, rectángulos, puntos, líneas y otras (cada clase de figura se deriva de la clase base **`Figura`**). Un administrador de pantalla utiliza apuntadores o referencias de la clase base (hacia **`Figura`**) para administrar todos los objetos a desplegar. Para dibujar cualquier objeto (independientemente del nivel en el que aparezca ese objeto en la jerarquía de herencia), el administrador de pantalla utiliza un apuntador de clase base (o referencia) hacia el objeto, y simplemente envía un mensaje **`dibujar`** hacia él. La función **`dibujar`** se declaró como virtual pura en la clase base **`Figura`** y se ignoró en cada una de las clases derivadas. Cada objeto de **`Figura`** sabe cómo dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse por el tipo de cada objeto, o de si el objeto es de un tipo que ha visto antes; el administrador de pantalla simplemente le dice a cada objeto que se dibuje a sí mismo.

El polimorfismo es particularmente efectivo para implementar sistemas de software en capas o niveles. Por ejemplo, en los sistemas operativos, cada tipo de dispositivo físico puede funcionar de manera diferente a los otros. Independientemente de esto, los comandos para *leer* o *escribir* datos desde y hacia dispositivos pueden tener cierta uniformidad. El mensaje *escribir* enviado a un objeto controlado por un dispositivo necesita interpretarse específicamente en el contexto de ese controlador de dispositivo, y en cómo es que ese controlador manipula los dispositivos de un tipo específico. Sin embargo, la llamada a *escribir* misma en realidad no es diferente de *escribir* para cualquier otro dispositivo del sistema; ésta simplemente coloca algunos bytes de la memoria en ese dispositivo. Un sistema operativo orientado a objetos puede utilizar una clase base abstracta para proporcionar una interfaz adecuada para todos los controladores de dispositivos. Entonces, a través de la herencia de esa clase base abstracta, las clases derivadas se forman para que todas funcionen de manera similar. Las capacidades (es decir, la interfaz pública) ofrecida por los controladores de dispositivos se proporcionan

como funciones virtuales puras en la clase base abstracta. Las implementaciones de estas funciones virtuales se proporcionan en las clases derivadas que corresponden a los tipos específicos de los controladores de dispositivos.

Con la programación polimórfica, un programa podría recorrer un contenedor tal como un arreglo de apuntadores a objetos, desde varios niveles de una jerarquía de clase. Los apuntadores de dicho arreglo serían apuntadores de la clase base hacia objetos de la clase derivada. Por ejemplo, un arreglo de objetos de la clase **`FigurasBidimensionales`** podría contener apuntadores **`FiguraBidimensional*`** hacia objetos de las clases derivadas **`Cuadrado`**, **`Circulo`**, **`Triangulo`**, **`Rectangulo`**, **`Linea`**, etcétera. Enviar un mensaje para dibujar cada objeto del arreglo, por medio del polimorfismo, dibujaría la imagen correcta en la pantalla.

## 20.6 Nuevas clases y vinculación dinámica

El polimorfismo y las funciones virtuales funcionan bastante bien cuando no se conoce por adelantado a todas las clases posibles. Sin embargo, también funcionan cuando se agregan nuevos tipos de clases a los sistemas.

Las nuevas clases se alojan por medio de la *vinculación dinámica* (también conocida como *vinculación tardía*). El tipo de un objeto no necesita conocerse en tiempo de compilación, para que se compile una llamada a una función **`virtual`**. En tiempo de ejecución, la llamada a la función **`virtual`** se hace coincidir con la función miembro del objeto llamado.

Un programa de administración de pantalla puede ahora desplegar nuevos tipos de objetos conforme se agregan al sistema, sin la necesidad de recompilar el administrador de pantalla. La llamada a la función dibujar permanece igual. Los nuevos objetos mismos contienen las capacidades reales de dibujo. Esto facilita la adición de nuevas capacidades a los sistemas con el mínimo impacto; también promueve la reutilización de software.

La vinculación dinámica permite a los fabricantes independientes de software distribuir su software sin tener que revelar los secretos del propietario. Las distribuciones de software pueden consistir solamente en archivos de encabezado y en archivos de objetos. No es necesario que se revele el código fuente. Entonces, los desarrolladores de software pueden utilizar la herencia para derivar nuevas clases, a partir de las proporcionadas por los fabricantes independientes. El software que funciona con las clases proporcionadas por dichos fabricantes, continuarán funcionando con las clases derivadas, y utilizarán (a través de la vinculación dinámica) las funciones virtuales sustituidas que se proporcionan en estas clases.

En la sección 20.8, presentamos un ejemplo práctico completo sobre polimorfismo. En la sección 20.9, describimos con detalle cómo se implementa en C++ el polimorfismo, las funciones virtuales y la vinculación dinámica.

## 20.7 Destructores virtuales

Cuando se utiliza el polimorfismo para procesar objetos asignados de una manera dinámica a una jerarquía de clase, puede ocurrir un problema. Si un objeto (con un destructor no **`virtual`**) se destruye explícitamente, aplicando el operador delete a un apuntador de clase base hacia el objeto, se llama a la función destructora de clase base (que coincida con el tipo del apuntador) sobre el objeto. Esto ocurre independiente del tipo del objeto al que apunta el apuntador de clase base, e independiente del hecho de que el destructor de cada clase tiene un nombre diferente.

Existe una solución sencilla para este problema; declare un destructor de clase base **`virtual`**. Esto hace que todos los destructores de clases derivadas sean virtuales, aunque no tengan el mismo nombre que el destructor de clase base. Ahora, si se destruye explícitamente a un objeto de la jerarquía, aplicando el operador delete a un apuntador de clase base que apunta hacia un objeto de clase derivada, se llama al destructor de la clase apropiada. Recuerde, cuando se destruye un objeto de clase derivada, la parte de la clase base correspondiente al objeto de la clase derivada también se destruye; el destructor de clase base siempre se ejecuta después del destructor de clase derivada.



**Buena práctica de programación 20.2**

*Si una clase tiene funciones virtuales, proporcione un destructor* **`virtual`***, incluso si no se necesita uno para la clase. Las clases derivadas de este tipo pueden contener destructores que deben invocarse adecuadamente.*



**Error común de programación 20.2**

*Los constructores no pueden ser virtuales. Declarar un constructor como una función* **virtual**, *es un error de sintaxis.*

## 20.8 Ejemplo práctico: Herencia de interfaz y de implementación

Nuestro siguiente ejemplo (figura 20.1) reexamina la jerarquía de **Punto**, **Circulo**, **Cilindro**, con la excepción de que ahora encabezamos la jerarquía con la clase base abstracta **Figura**. **Figura** tiene dos funciones virtuales puras, **imprimeNombreFigura** e **imprime**, de tal modo que es una clase base abstracta. **Figura** contiene otras dos funciones virtuales, **area** y **volumen**, cada una de las cuales tiene una implementación predeterminada que devuelve un valor de cero. **Punto** hereda estas implementaciones de **Figura**. Esto tiene sentido, ya que el área y el volumen de un punto son cero. **Circulo** hereda la función **volumen** de **Punto**, pero proporciona su propia implementación para la función **area**. **Cilindro** proporciona sus propias implementaciones tanto para la función **area** como para la función **volumen**.

```
// Figura 20.1: figura.h
// Definición de la clase base abstracta Figura
#ifndef FIGURA_H
#define FIGURA_H

class Figura {
public:
   virtual double area() const { return 0.0; }
   virtual double volumen() const { return 0.0; }

   // funciones virtuales puras sustituidas en clases derivadas
   virtual void imprimeNombreFigura() const = 0;
   virtual void imprime() const = 0;
}; // fin de la clase Figura

#endif
```

Figura 20.1 Demostración de la herencia de interfaz con la jerarquía de la clase **Figura**; **figura.h**.

```
// Figura 20.1: punto1.h
// Definición de la clase Punto
#ifndef PUNTO1_H
#define PUNTO1_H

#include <iostream>

using std::cout;

#include “figura.h”

class Punto : public Figura {
public:
   Punto( int = 0, int = 0 );  // constructor predeterminado
   void establecePunto( int, int );
   int obtieneX() const { return x; }
   int obtieneY() const { return y; }
```

Figura 20.1 Demostración de la herencia de interfaz con la jerarquía de la clase **Figura**; **punto1.h**. (Parte 1 de 2.)

```
   virtual void imprimeNombreFigura() const { cout << “Punto: “; }
   virtual void imprime() const;
private:
   int x, y;   // coordenadas x e y de Punto
}; // fin de la clase Punto

#endif
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`punto1.h`**. (Parte 2 de 2.)

```
// Figura 20.1: punto1.cpp
// Definición de las funciones miembro para la clase Punto
#include “punto1.h”

Punto::Punto( int a, int b ) { establecePunto( a, b ); }

void Punto::establecePunto( int a, int b )
{
   x = a;
   y = b;
} // fin de la función establecePunto

void Punto::imprime() const
   { cout << ‘[‘ << x << “, “ << y << ‘]’; }
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`punto1.cpp`**.

```
// Figura 20.1: circulo1.h
// Definición de la clase Circulo
#ifndef CIRCULO1_H
#define CIRCULO1_H
#include “punto1.h”

class Circulo : public Punto {
public:
   // constructor predeterminado
   Circulo( double r = 0.0, int x = 0, int y = 0 );

   void estableceRadio( double );
   double obtieneRadio() const;
   virtual double area() const;
   virtual void imprimeNombreFigura() const { cout << “Circulo: “; }
   virtual void imprime() const;
private:
   double radio;   // radio del Circulo
}; // fin de la clase Circulo

#endif
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`circulo1.h`**.

```
// Figura 20.1: circulo1.cpp
// Definición de las funciones miembro para la clase Circulo
#include <iostream>

using std::cout;

#include “circulo1.h”

Circulo::Circulo( double r, int a, int b )
   : Punto( a, b )  // llama al constructor de la clase base
{ estableceRadio( r ); }

void Circulo::estableceRadio( double r ) { radio = r > 0 ? r : 0; }

double Circulo::obtieneRadio() const { return radio; }

double Circulo::area() const
   { return 3.14159 * radio * radio; }

void Circulo::imprime() const
{
   Punto::imprime();
   cout << “; Radio = “ << radio;
} // fin de la función imprime
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`circulo1.cpp`**.

```
// Figura 20.1: cilindro1.h
// Definición de la clase Cilindro
#ifndef CILINDRO1_H
#define CILINDRO1_H
#include “circulo1.h”

class Cilindro : public Circulo {
public:
   // constructor predeterminado
   Cilindro( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void estableceAltura( double );
   double obtieneAltura();
   virtual double area() const;
   virtual double volumen() const;
   virtual void imprimeNombreFigura() const {cout << “Cilindro: “;}
   virtual void imprime() const;
private:
   double altura;   // altura del Cilindro
}; // fin de la clase Cilindro

#endif
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`cilindro1.h`**.

```
// Figura 20.1: cilindro1.cpp
// Definición de las funciones y amigas para la clase Cilindro
#include <iostream>

using std::cout;

#include “cilindro1.h”

Cilindro::Cilindro( double h, double r, int x, int y )
   : Circulo( r, x, y )  // llama al constructor de la clase base
{ estableceAltura( h ); }

void Cilindro::estableceAltura( double h )
   { altura = h > 0 ? h : 0; }

double Cilindro::obtieneAltura() { return altura; }

double Cilindro::area() const
{
   // superficie del Cilindro
   return 2 * Circulo::area() +
          2 * 3.14159 * obtieneRadio() * altura;
} // fin de la función área

double Cilindro::volumen() const
   { return Circulo::area() * altura; }

void Cilindro::imprime() const
{
   Circulo::imprime();
   cout << “; Altura = “ << altura;
} // fin de la función imprime
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`cilindro1.cpp`**.

```
// Figura 20.1: fig20_01.cpp
// Controlador para la jerarquía figura, punto, circulo, cilindro
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include “figura.h”
#include “punto1.h”
#include “circulo1.h”
#include “cilindro1.h”

void apuntadorViaVirtual( const Figura * );
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`fig20_01.cpp`**. (Parte 1 de 3.)

```
void referenciaViaVirtual( const Figura & );

int main()
{
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Punto punto( 7, 11 );                  // crea Punto
   Circulo circulo( 3.5, 22, 8 );         // crea Circulo
   Cilindro cilindro( 10, 3.3, 10, 10 );  // crea Cilindro

   punto.imprimeNombreFigura();    // vinculación estática
   punto.imprime();                // vinculación estática
   cout << ‘\n’;

   circulo.imprimeNombreFigura();  // vinculación estática
   circulo.imprime();              // vinculación estática
   cout << ‘\n’;

   cilindro.imprimeNombreFigura(); // vinculación estática
   cilindro.imprime();             // vinculación estática
   cout << “\n\n”;

   Figura *arregloDeFiguras[ 3 ]; // arreglo de apuntadores a la clase base

   // arregloDeFiguras[0] apunta al objeto Punto de la clase derivada
   arregloDeFiguras[ 0 ] = &punto;

   // arregloDeFiguras[1] apunta al objeto Circulo de la clase derivada
   arregloDeFiguras[ 1 ] = &circulo;

   // arregloDeFiguras[2] apunta al objeto Cilindro de la clase derivada
   arregloDeFiguras[ 2 ] = &cilindro;

   // Ciclo a través de arregloDeFiguras y llamada a apuntadorViaVirtual
   // para imprimir el nombre de la forma, atributos, area, y volumen
   // de cada objeto mediante vinculación dinámica.
   cout << “Llamadas virtuales a funciones mediante “
        << “apuntadores a la clase base\n”;

   for ( int i = 0; i < 3; i++ )
      apuntadorViaVirtual( arregloDeFiguras[ i ] );

   // Ciclo a través de arregloDeFiguras y llamada a referenciaViaVirtual
   // para imprimir el nombre de la forma, atributos, area, y volumen
   // de cada objeto mediante vinculación dinámica.
   cout << “Llamadas virtuales a funciones mediante “
        << “referencias a la clase base\n”;

   for ( int j = 0; j < 3; j++ )
      referenciaViaVirtual( *arregloDeFiguras[ j ] );

   return 0;
} // fin de la función main

```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **Figura**; **fig20_01.cpp**. (Parte 2 de 3.)

```
// Hace llamada llamadas a funciones virtuales mediante un apuntador a la clase base
// con el uso de vinculación estática
void apuntadorViaVirtual( const Figura *ptrClaseBase )
{
   ptrClaseBase->imprimeNombreFigura();
   ptrClaseBase->imprime();
   cout << “\nArea = “ << ptrClaseBase->area()
        << “\nVolumen = “ << ptrClaseBase->volumen() << “\n\n”;
} // fin de la función apuntadorViaVirtual

// Hace llamada llamadas a funciones virtuales mediante una referencia a la clase base
// con el uso de vinculación estática.
void referenciaViaVirtual( const Figura &refClaseBase )
{
   refClaseBase.imprimeNombreFigura();
   refClaseBase.imprime();
   cout << “\nArea = “ << refClaseBase.area()
        << “\nVolumen = “ << refClaseBase.volumen() << “\n\n”;
} // fin de la función referenciaViaVirtual
```

```
Punto: [7, 11]
Circulo: [22, 8]; Radio = 3.50
Cilindro: [10, 10]; Radio = 3.30; Altura = 10.00

Llamadas virtuales a funciones mediante apuntadores a la clase base
Punto: [7, 11]
Area = 0.00
Volumen = 0.00

Circulo: [22, 8]; Radio = 3.50
Area = 38.48
Volumen = 0.00

Cilindro: [10, 10]; Radio = 3.30; Altura = 10.00
Area = 275.77
Volumen = 342.12

Llamadas virtuales a funciones mediante referencias a la clase base
Punto: [7, 11]
Area = 0.00
Volume = 0.00

Circulo: [22, 8]; Radio = 3.50
Area = 38.48
Volumen = 0.00

Cilindro: [10, 10]; Radio = 3.30; Altura = 10.00
Area = 275.77
Volumen = 342.12
```

**Figura 20.1** Demostración de la herencia de interfaz con la jerarquía de la clase **`Figura`**; **`fig20_01.cpp`**. (Parte 3 de 3.)

Observe que aunque **`Figura`** es una clase base abstracta, aún contiene implementaciones de ciertas funciones miembro, y que dichas implementaciones son heredables. La clase **`Figura`** proporciona una interfaz heredable en forma de cuatro funciones virtuales que contendrán todos los miembros de la jerarquía. La clase

**Figura** también proporciona algunas implementaciones que utilizarán las clases derivadas en los primeros niveles de la jerarquía.



### Observación de ingeniería de software 20.8

*Una clase puede heredar la interfaz y/o la implementación de una clase. Las jerarquías diseñadas para la herencia de implementaciones tienden a tener su funcionalidad más arriba en la jerarquía; cada nueva clase derivada hereda una o más de las funciones miembro que se definieron en una clase base, y la nueva clase derivada utiliza las definiciones de la clase base. Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad más abajo en la jerarquía; una clase base especifica una o más funciones que deben definirse para cada clase de la jerarquía (es decir, tienen la misma firma), pero las clases derivadas individuales proporcionan sus propias implementaciones de funciones.*

La clase base **Figura** (líneas 1 a 16) consiste en cuatro funciones **public virtual** y no contiene dato alguno. Las funciones **imprimeNombreFigura** e **imprime** son virtuales puras, por lo se pasan por alto en cada una de las clases derivadas. Estas funciones se pasan por alto en las clases derivadas, cuando es apropiado que esas clases tengan un cálculo diferente de área y/o de volumen. Observe que **Figura** es una clase abstracta y que contiene algunas funciones virtuales "impuras" (**area** y **volumen**). Las clases abstractas también pueden incluir funciones y datos no virtuales, los cuales serán heredados por las clases derivadas.

La clase **Punto** (líneas 17 a 54) se deriva de **Figura** con herencia pública. Un punto tiene un área de 0.0 y un volumen de 0.0, por lo que las funciones miembro de clase base **area** y volumen aquí no se pasan por alto; éstas simplemente son heredadas como están definidas en **Figura**. Las funciones **imprimeNombreFigura** e **imprime** son implementaciones de funciones virtuales que se definieron como puras en la clase base; si no pasamos por alto a estas funciones en la clase **Punto**, entonces **Punto** también sería una clase abstracta, y no podríamos instanciar a los objetos de **Punto**. Otras funciones miembro incluyen una función *establecer* para asignar nuevas coordenadas **x** y **y** a un **Punto**, y funciones *obtener* para devolver las coordenadas **x** e **y** de un **Punto**.

La clase **Circulo** (líneas 55 a 99) se deriva de **Punto** con herencia pública. Un círculo tiene un volumen de 0.0, por lo que la función miembro de clase base volumen aquí no se pasan por alto; ésta se hereda desde **Punto**, la cual heredó volumen desde **Figura**. Un **Circulo** tiene un área diferente de cero, por lo que la función **area** se pasa por alto en esta clase. Las funciones **imprimeNombreFigura** e **imprime** son implementaciones de funciones virtuales que se definieron como puras en la clase **Figura**. Si estas funciones no se pasan por alto aquí, las versiones de **Punto** correspondientes a estas funciones se heredarían. Otras funciones miembro incluyen una función *establecer* para asignar un nuevo radio a un **Circulo**, y una función *obtener* para devolver el radio de un **Circulo**.

La clase **Cilindro** (líneas 100 a 154) se deriva de **Circulo** con herencia pública. Un cilindro tiene un área y un volumen diferentes a los de **Circulo**, por lo que en esta clase se pasan por alto tanto la función **area** como la función volumen. Las funciones **imprimeNombreFigura** e **imprime** son implementaciones de funciones virtuales que se definieron como puras en la clase **Figura**. Si estas funciones no se pasan por alto aquí, las versiones de **Circulo** correspondientes a estas funciones se heredarían. Otras funciones miembro incluyen funciones *establecer* y *obtener* para asignar una nueva altura y para devolver la altura de un **Cilindro**, respectivamente.

El programa controlador (líneas 155 a 247) comienza creando una instancia del objeto punto correspondiente a **Punto**, del objeto circulo de **Circulo** y del objeto cilindro de **Cilindro**. Las funciones **imprimeNombreFigura** e **imprime** se invocan para cada objeto, para que impriman el nombre de cada uno de ellos y para mostrar que los objetos se inicializan correctamente. Cada llamada a las funciones **imprimeNombreFigura** e **imprime** de las líneas 185 a 194 utiliza una vinculación estática; en tiempo de compilación, el compilador conoce el tipo de cada objeto para los que se invocó a las funciones **imprimeNombreFigura** e **imprime**.

Después, se declara el arreglo **arregloDeFiguras**, cuyos elementos son del tipo **Figura***. Este arreglo de apuntadores de clase base se utiliza para apuntar a cada uno de los objetos de clase derivada. La dirección del objeto **punto** se asigna a **arregloDeFiguras[ 0 ]** (línea 200), la dirección del objeto **circulo** se asigna a **arregloDeFiguras[ 1 ]** (línea 203), y la dirección del objeto cilindro se asigna a **arregloDeFiguras[ 2 ]** (línea 206).

Posteriormente, una estructura **for** (línea 214) recorre el arreglo **arregloDeFiguras**, e invoca a la función **apuntadorViaVirtual** (línea 215)

```
apuntadorViaVirtual( arregloDeFiguras[ i ] );
```

para cada elemento del arreglo. La función **apuntadorViaVirtual** recibe en el parámetro **ptrClaseBase** (de tipo **const Figura ***) la dirección almacenada en un elemento de **arregloDeFiguras**. Cada vez que se ejecuta **apuntadorViaVirtual**, se realizan las siguientes cuatro llamadas de función virtual

```
ptrClaseBase->imprimeNombreFigura()
ptrClaseBase->imprime()
ptrClaseBase->area()
ptrClaseBase->volumen()
```

Cada una de estas llamadas invocan a una función **virtual** sobre el objeto al que apunta **ptrClaseBase** en tiempo de ejecución; un objeto cuyo tipo no puede determinarse aquí en tiempo de compilación. La salida ilustra que se invoca a las funciones apropiadas para cada clase. Primero, se despliegan la cadena **"Punto: "**, y las coordenadas del objeto **punto**; el área y el volumen son 0.0. Después, se despliegan la cadena **"Circulo: "**, y las coordenadas del centro del objeto **circulo** y el radio del objeto **circulo**; el área del círculo se calcula y el volumen se devuelve como 0.0. Por último, se despliegan la cadena **"Cilindro: "**, las coordenadas del centro de la base del objeto **cilindro**, el radio y la altura del objeto **cilindro**; el área y el volumen del **cilindro** se calculan. Todas las llamadas a funciones virtual **imprimeNombreFigura**, **imprime**, **area**, y **volumen** se resuelven en tiempo de ejecución por medio de la vinculación dinámica.

Por último, una estructura **for** (línea 223) recorre **arregloDeFiguras** e invoca a la función **referenciaViaVirtual** (línea 224)

```
referenciaViaVirtual( *arregloDeFiguras[ j ] );
```

para cada elemento del arreglo. La función **referenciaViaVirtual** recibe en su parámetro a **refClaseBase** (de tipo **const Figura&**), una referencia que se formada al desreferenciar la dirección almacenada en un elemento del arreglo. Durante cada llamada a **referenciaViaVirtual**, se realizan las siguientes llamadas de función virtual

```
refClaseBase.imprimeNombreFigura()
refClaseBase.imprime()
refClaseBase.area()
refClaseBase.volumen()
```

Cada una de las llamadas anteriores invoca a estas funciones sobre el objeto al que se refiere **refClaseBase**. La salida producida utilizando referencias de clase base, es idéntica a la salida producida utilizando apuntadores de clase base.

## 20.9 Polimorfismo, funciones virtuales y vinculación dinámica "tras bambalinas"

C++ hace que el polimorfismo sea fácil de programar. Es cierto que es posible programar el polimorfismo en lenguajes no orientados a objetos como C, pero para hacerlo se requieren manipulaciones de apuntadores complejas y potencialmente peligrosas. En esta sección explicamos cómo es que C++ implementa internamente el polimorfismo, las funciones virtuales y la vinculación dinámica. Esto le proporcionará una comprensión sólida de cómo es que en realidad funcionan estas capacidades. Aún más importante, le ayudará a apreciar la sobrecarga del polimorfismo, con respecto al consumo de memoria y al tiempo de procesamiento. Esto le ayudará a determinar cuándo utilizar el polimorfismo, y cuándo evitarlo.

Primero explicaremos las estructuras de datos que el compilador de C++ construye en tiempo de compilación para soportar el polimorfismo en tiempo de ejecución. Después mostraremos cómo un programa en ejecución utiliza estas estructuras de datos para ejecutar funciones virtuales y para lograr la vinculación dinámica asociada con el polimorfismo.

Cuando C++ compila una clase que tiene una o más funciones virtuales, éste construye una *tabla de funciones virtuales* (*vtable*) para esa clase. El programa en ejecución utiliza la *vtable* para seleccionar las implementaciones de la función apropiada, cada vez que va a ejecutarse una función **virtual** de esa clase. La figura 20.2 muestra las tablas de funciones virtuales para las clases **Figura**, **Punto**, **Circulo** y **Cilindro**.

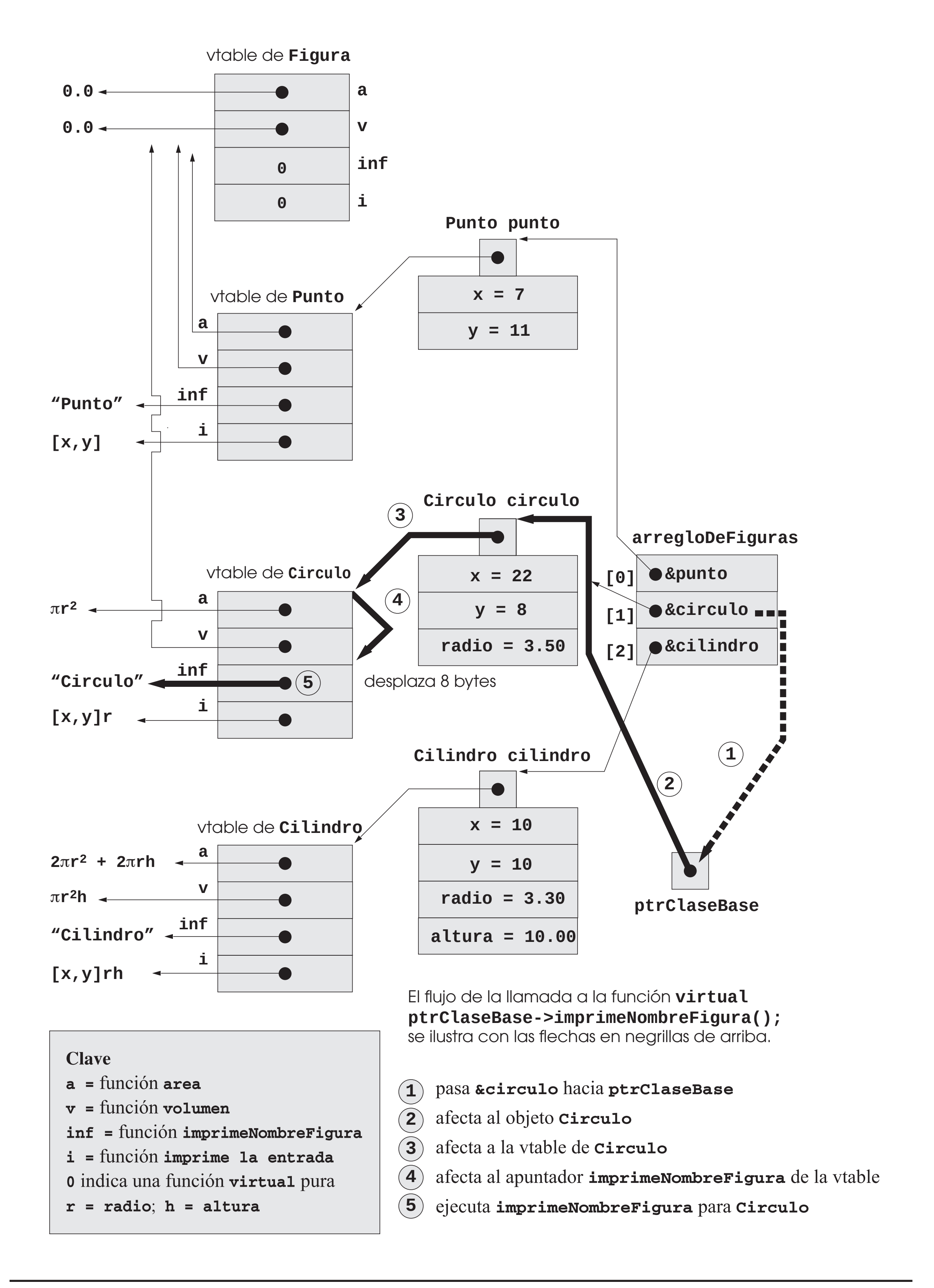



**Figura 20.2** Flujo de control de una llamada a una función `virtual`.

En la *vtable* de la clase **Figura**, el primer apuntador de función apunta a la implementación de la función **area** para esa clase, a saber, una función que devuelve un área de 0.0. El segundo apuntador apunta hacia el volumen, una función que también devuelve 0.0. Las funciones **imprimeNombreFigura** e **imprime** son virtuales puras; ellas carecen de implementaciones, por lo que sus apuntadores de función se establecen en 0. Cualquier clase que tenga uno o más apuntadores 0 en su *vtable* es una clase abstracta. Las clases sin apuntadores 0 en su *vtable* (como **Punto**, **Circulo** y **Cilindro**) son clases concretas.

La clase **Punto** hereda las funciones **area** y **volumen** de la clase **Figura**, por lo que el compilador simplemente establece estos dos apuntadores en la *vtable* para la clase **Punto**, para que sean copias de los apuntadores **area** y **volumen** de la clase **Figura**. La clase **Punto** pasa por alto la función **imprimeNombreFigura** para imprimir **"Punto: "**, por lo que la función apuntador apunta hacia la función **imprimeNombreFigura** de la clase **Punto**. **Punto** también pasa por alto la función imprime, por lo que la función apuntador correspondiente apunta hacia la función de la clase **Punto** que imprime **[x, y]**.

El apuntador de la función **area** de **Circulo** que se encuentra en la *vtable* para la clase **Circulo**, apunta hacia la función **area** de **Circulo** que devuelve $\pi r^2$. El apuntador de la función volumen simplemente se copia desde la clase **Punto**; ese apuntador se copió previamente en **Punto** desde **Figura**. El apuntador de la función **imprimeNombreFigura** apunta hacia la versión **Circulo** de la función que imprime **"Circulo: "**. El apuntador de la función **imprime** apunta hacia la función **imprime** de **Circulo** que imprime **[x, y]r**.

El apuntador de la función **area** de la *vtable* para la clase **Cilindro** apunta hacia la función **area** de **Cilindro** que calcula la superficie del **Cilindro**, a saber $2\pi r^2 + 2\pi rh$. El apuntador de la función volumen correspondiente a **Cilindro** apunta hacia una función volumen que devuelve $\pi r^2 h$. El apuntador de la función **imprimeNombreFigura** correspondiente a **Cilindro** apunta hacia una función que imprime **"Cilindro: "**. El apuntador de la función **imprime** correspondiente a **Cilindro** apunta hacia su función que imprime **[x, y]rh**.

El polimorfismo se logra a través de una compleja estructura de datos que involucra tres niveles de apuntadores. Hemos explicado un nivel: los apuntadores a las funciones en la *vtable*. Estos apuntadores apuntan hacia las funciones reales a ejecutarse cuando se invoca a una función **virtual**.

Ahora consideremos el segundo nivel de apuntadores. Siempre que se instancia un objeto de una clase con funciones virtuales, el compilador adjunta al frente del objeto un apuntador hacia la *vtable* para esa clase. [*Nota:* Este apuntador normalmente está al frente del objeto, pero no se requiere que se implemente de esa manera.]

El tercer nivel de apuntadores es simplemente el manejo del objeto que está recibiendo la llamada a la función **virtual** (este manejo también puede ser una referencia).

Ahora veamos cómo se ejecuta una llamada a una función **virtual** típica. Considere la llamada

```
ptrClaseBase->imprimeNombreFigura()
```

en la función **apuntadorViaVirtual**. Suponga para la siguiente explicación que **ptrClaseBase** contiene la dirección de **arregloDeFiguras[ 1 ]** (es decir, la dirección del objeto **circulo**). Cuando el compilador compila esta instrucción, determina que en realidad la llamada la está realizando un apuntador de clase base, y que **imprimeNombreFigura** es una función **virtual**.

Después, el compilador determina que **imprimeNombreFigura** es la tercera entrada en cada una de las *vtables*. Para localizar esta entrada, el compilador observa que necesitará ignorar las dos primeras entradas. Entonces, el compilador compila un *desplazamiento* de 8 bytes (4 bytes para cada apuntador en las máquinas de 32 bits actuales) en el código del objeto en lenguaje máquina que ejecutará la llamada a la función **virtual**.

Posteriormente, el compilador genera código que hará lo siguiente [*Nota:* Los números de la lista de abajo corresponden a los números encerrados en círculos de la figura 20.2]:

1. Seleccionará la *iésima* entrada del **arregloDeFiguras** (en este caso la dirección del objeto **circulo**), y lo pasará hacia el **apuntadorViaVirtual**. Esto establece al **ptrClaseBase** para que apunte hacia **circulo**.
2. Desreferenciará ese apuntador para afectar al objeto **circulo**, el cual, como usted recordará, comienza con un apuntador hacia la *vtable* de **Circulo**.
3. Desreferenciará el apuntador de la *vtable* de **circulo** para afectar a la *vtable* de **Circulo**.

4. Ignorará el desplazamiento de 8 bytes para recoger el apuntador de la función **`imprimeNombre-Figura`**.
5. Desreferenciará al apuntador de la función **`imprimeNombreFigura`** para formar el nombre de la función real a ejecutarse, y utilizará el operador de llamada a una función **`()`** para ejecutar la función **`imprimeNombreFigura`** adecuada e imprimirá la cadena de caracteres **“`Circulo: `”**.

Las estructuras de datos de la figura 20.2 pueden parecer complejas, pero la mayor parte de esta complejidad es manejada por el compilador y está oculta al programador, lo que hace que la programación polimórfica en C++ sea directa.

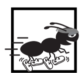
Las operaciones para desreferenciar un apuntador y los accesos a la memoria que ocurren en toda llamada a una función **`virtual`** requieren algo de tiempo de ejecución adicional. Las *vtable´s* y los apuntadores a las *vtable*s agregadas a los objetos requieren algo de memoria adicional.

Ojalá ahora tenga suficiente información sobre cómo operan las funciones virtuales, para que determine si es apropiado utilizarlas en cada aplicación que considere.

**Tip de rendimiento 20.1**

*El polimorfismo es eficiente, mientras se implemente con funciones virtuales y vinculación dinámica. Los programadores pueden utilizar estas capacidades con un efecto nominal en el rendimiento del sistema.*

**Tip de rendimiento 20.2**



*Las funciones virtuales y la vinculación dinámica permiten que la programación polimórfica sea el opuesto de la programación con switch lógicos. Los compiladores de C++ normalmente generan código que se ejecuta al menos tan eficientemente que el código manual basado en switch lógicos. De una u otra forma, la sobrecarga del polimorfismo es aceptable para la mayoría de las aplicaciones. Sin embargo, en algunas situaciones (por ejemplo, en aplicaciones en tiempo real con requerimientos rigurosos de rendimiento), la sobrecarga del polimorfismo puede ser muy alta.*

## *RESUMEN*

- Con las funciones virtuales y el polimorfismo, se hace posible diseñar e implementar sistemas que sean más fácilmente extensibles. Los programas pueden escribirse para procesar objetos de tipos que pueden no existir cuando el programa está en desarrollo.
- La programación polimórfica con funciones virtuales puede eliminar la necesidad del switch lógico. El programador puede utilizar el mecanismo de una función **`virtual`** para desarrollar la lógica equivalente, con lo que se evitan los tipos de errores generalmente asociados con el switch lógico. El código cliente que toma decisiones sobre los tipos de objetos y las representaciones indica un diseño de clase pobre.
- Si es necesario, las clases derivadas pueden proporcionar sus propias implementaciones de una función **`virtual`** de clase base, pero si no lo es, se utiliza la implementación de la clase base.
- Si se llama a una función **`virtual`**, haciendo referencia a un objeto específico por su nombre y utilizando el operador punto de selección de miembro, la referencia se resuelve en tiempo de compilación (a esto se le conoce como vinculación estática), y la función **`virtual`** que es llamada es la definida (o heredada) por la clase de ese objeto en particular.
- Existen muchas situaciones en las que es útil definir clases para las que el programador nunca intenta crear instancias de ningún objeto. Dichas clases se conocen como clases abstractas. Éstas se utilizan sólo como clases base, por lo que normalmente no referiremos a ellas como clases base abstractas. Ningún objeto de una clase abstracta puede instanciarse en un programa.
- Las clases cuyos objetos pueden instanciarse se conocen como clases concretas.
- Una clase se hace abstracta, declarando una o más funciones virtuales como puras. Una función **`virtual`** pura es aquella que tiene un inicializador =0 en su declaración.
- Si una clase se deriva de una clase con una función **`virtual`** pura, sin suplir la definición de esa función **`virtual`** pura en la clase derivada, entonces esa función **`virtual`** permanece pura en la clase derivada. Como consecuencia, la clase derivada también es una clase abstracta.
- C++ permite el polimorfismo; la habilidad de los objetos de diferentes clases relacionadas por la herencia de responder de manera diferente a la misma llamada a la función miembro.

- El polimorfismo se implementa a través de funciones virtuales.
- Cuando se hace una solicitud a través de un apuntador de clase base o referencia para utilizar una función **`virtual`**, C++ elige la función correcta en la clase derivada asociada con el objeto.
- Por medio de las funciones virtuales y el polimorfismo, una llamada a una función miembro puede ocasionar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada.
- Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar apuntadores hacia ellas. Tales apuntadores pueden utilizarse para permitir manipulaciones polimórficas de objetos de clases derivadas, cuando dichos objetos se instancian a partir de clases concretas.
- Por lo general, nuevos tipos de clases se añaden a los sistemas. Las nuevas clases son alojadas por medio de la vinculación dinámica (también conocida como vinculación tardía). El tipo de un objeto no necesita conocerse en tiempo de compilación, para que una llamada a una función **`virtual`** se compile. En tiempo de ejecución, se hace que la llamada a la función **`virtual`** coincida con la función miembro del objeto que la recibe.
- La vinculación dinámica permite a los fabricantes de software independientes distribuir software sin revelar secretos del propietario. Las distribuciones de software pueden consistir solamente en archivos de encabezado y en archivos de objetos. No es necesario revelar el código fuente. Los desarrolladores de software pueden entonces utilizar la herencia para derivar nuevas clases a partir de aquellas provistas por los fabricantes. El software que funciona con las clases de los fabricantes independientes de software continuará funcionando con las clases derivadas, y utilizara (a través de la vinculación dinámica) las funciones sustituidas provistas en estas clases.
- La vinculación dinámica requiere que, en tiempo de ejecución, la llamada a la función miembro **`virtual`** se enrute hacia la versión de la función **`virtual`** apropiada para la clase. Una tabla de funciones virtual llamada *vtable* se implementa como un arreglo que contiene apuntadores a las funciones. Cada clase con funciones **`virtual`** tiene una *vtable*. Para cada función **`virtual`** en la clase, la *tablav* tiene una entrada que contiene un apuntador de función hacia la versión de la función **`virtual`** a utilizar para un objeto de esa clase. La función **`virtual`** a utilizar para una clase en particular podría ser la función definida en esa clase, o podría ser una función heredada directa o indirectamente desde una clase base más arriba en la jerarquía.
- Cuando una clase base proporciona una función miembro **`virtual`**, las clases derivadas pueden pasar por alto a la función **`virtual`**, pero no tienen que hacerlo. Entonces, una clase derivada puede utilizar una versión de una clase base correspondiente a una función miembro, y esto se indicaría en la *vtable*.
- Cada objeto de una clase con funciones **`virtual`** contiene un apuntador a la *vtable* para esa clase. El apuntador de la función adecuada en la *vtable* se obtiene y se desreferencia para completar la llamada en tiempo de ejecución. Esta búsqueda en la *vtable* y la desreferencia de un apuntador requieren una sobrecarga nominal en tiempo de ejecución, normalmente menor que el mejor código cliente.
- Declara el destructor de la clase como **`virtual`**, si la clase contiene funciones virtuales. Esto hace que todos los destructores de clases derivadas sean virtuales, aunque no tengan el mismo nombre que el destructor de la clase base. Si un objeto de la jerarquía se destruye explícitamente, aplicando el operador delete a un apuntador de clase base hacia un objeto de clase derivada, se llama al destructor de la clase apropiada.
- Cualquier clase que tenga uno o más apuntadores 0 en su *vtable*, es una clase abstracta. Las clases sin apuntadores 0 en la *vtable* (como **`Punto`**, **`Circulo`** y **`Cilindro`**), son clases concretas.

## *TERMINOLOGÍA*

apuntador hacia una clase abstracta
apuntador hacia una clase base
apuntador hacia una clase derivada
apuntador hacia una *vtable*
clase abstracta
clase base abstracta
clase concreta
clase derivada
constructor de clase derivada
conversión explícita de apuntadores
desplazamiento en una *vtable*
destructor **`virtual`**
eliminación de instrucciones **`switch`**
extensibilidad
fabricantes independientes de software
función **`virtual`**
función **`virtual`** de clase base
función **`virtual`** pura (=0)
herencia
herencia de implementación
herencia de interfaz
jerarquía de clase
pasar por alto a una función **`virtual`**
pasar por alto a una función **`virtual`** pura
polimorfismo
programación “en lo general”
programación “en lo particular”
referencia a una clase abstracta
referencia a una clase base
referencia a una clase derivada
reutilización de software
**`switch`** lógico
tabla de funciones virtuales
vinculación dinámica
vinculación estática
vinculación tardía
*vtable*

## ERRORES COMUNES DE PROGRAMACIÓN

**20.1** Intentar crear una instancia de un objeto correspondiente a una clase abstracta (es decir, una clase que contiene una o más funciones virtuales), es un error de sintaxis.

**20.2** Los constructores no pueden ser virtuales. Declarar un constructor como una función **`virtual`**, es un error de sintaxis.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**20.1** Aun cuando ciertas funciones son implícitamente virtuales, debido a una declaración hecha en un nivel superior de la jerarquía de la clase, declare explícitamente estas funciones como **`virtual`** en cada nivel de la jerarquía para promover la claridad del programa.

**20.2** Si una clase tiene funciones virtuales, proporcione un destructor **`virtual`**, incluso si no se necesita uno para la clase. Las clases derivadas de este tipo pueden contener destructores que deben invocarse adecuadamente.

## TIPS DE RENDIMIENTO

**20.1** El polimorfismo es eficiente, mientras se implemente con funciones virtuales y vinculación dinámica. Los programadores pueden utilizar estas capacidades con un efecto nominal en el rendimiento del sistema.

**20.2** Las funciones virtuales y la vinculación dinámica permiten que la programación polimórfica sea el opuesto de la programación con switch lógicos. Los compiladores de C++ normalmente generan código que se ejecuta al menos tan eficientemente que el código manual basado en switch lógicos. De una u otra forma, la sobrecarga del polimorfismo es aceptable para la mayoría de las aplicaciones. Sin embargo, en algunas situaciones (por ejemplo, en aplicaciones en tiempo real con requerimientos rigurosos de rendimiento), la sobrecarga del polimorfismo puede ser muy alta.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**20.1** Una consecuencia interesante de utilizar funciones virtuales y el polimorfismo es que los programas adquieren una apariencia simplificada. Éstos contienen menos divisiones lógicas, a favor de código secuencial más sencillo. Esto facilita la evaluación, la depuración y el mantenimiento de programas, así como la eliminación de errores.

**20.2** Una vez que una función se declara como virtual, ésta permanece así en todos los niveles inferiores de la jerarquía de herencia a partir de ese punto, incluso si no se le declara como virtual cuando una clase la sustituye.

**20.3** Cuando una clase derivada elige no definir una función **`virtual`**, la clase derivada simplemente hereda la definición de la función **`virtual`** de la clase base inmediata.

**20.4** Si una clase se deriva de una clase con una función **`virtual`** pura, y si no se proporciona una definición para dicha función en la clase derivada, entonces esa función virtual permanece pura en la clase derivada. En consecuencia, la clase derivada también es una clase abstracta.

**20.5** Con las funciones virtuales y el polimorfismo, el programador puede manejar generalidades y dejar que el ambiente en tiempo de ejecución se ocupe de las particularidades. El programador puede manejar una amplia variedad de objetos para que se comporten de manera apropiada, sin siquiera tener que conocer los tipos de esos objetos.

**20.6** El polimorfismo promueve la extensibilidad: el software escrito para invocar un comportamiento polimórfico se escribe de manera independiente de los tipos de los objetos a los que se envían los mensajes. Entonces, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en un sistema, sin tener que modificar el sistema base. Con excepción del código cliente que genera instancias de nuevos objetos, los programas no necesitan recompilarse.

**20.7** Una clase abstracta define una interfaz para los diferentes miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras que se definirán en las clases derivadas. Todas las funciones de la jerarquía pueden utilizar esta misma interfaz, a través del polimorfismo.

**20.8** Una clase puede heredar la interfaz y/o la implementación de una clase. Las jerarquías diseñadas para la *herencia de implementaciones* tienden a tener su funcionalidad más arriba en la jerarquía; cada nueva clase derivada hereda una o más de las funciones miembro que se definieron en una clase base, y la nueva clase derivada utiliza las definiciones de la clase base. Las jerarquías diseñadas para la *herencia de interfaz* tienden a tener su funcionalidad más abajo en la jerarquía; una clase base especifica una o más funciones que deben definirse para cada clase de la jerarquía (es decir, tienen la misma firma), pero las clases derivadas individuales proporcionan sus propias implementaciones de funciones.

## EJERCICIOS DE AUTOEVALUACIÓN

**20.1** Complete los espacios en blanco:
- a) Utilizar la herencia y el polimorfismo ayuda a eliminar el ____________ lógico.
- b) Una función **`virtual`** pura se especifica colocando ____________ al final de su prototipo en la definición de la clase.
- c) Si una clase contiene una o más funciones virtuales puras, se trata de una ____________.
- d) Una llamada a una función resuelta en tiempo de compilación se conoce como vinculación ____________.
- e) Una llamada a una función resuelta en tiempo de ejecución se conoce como vinculación ____________.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**20.1** a) **`switch`**. b) **`=0`**. c) Clase base abstracta. d) Estática. e) Dinámica o tardía.

## EJERCICIOS

**20.2** ¿Qué son las funciones virtuales? Describa una circunstancia en la que dichas funciones serían adecuadas.

**20.3** Dado que los constructores no pueden ser virtuales, describa un esquema en el que usted podría lograr un efecto similar.

**20.4** ¿Cómo es que el polimorfismo le permite programar “en lo general”, en lugar de hacerlo “en lo particular”. Explique las ventajas claves de la programación “en lo general”.

**20.5** Explique los problemas de la programación con **`switch`** lógico. Explique por qué el polimorfismo es una alternativa efectiva para el uso del **`switch`** lógico.

**20.6** Plantee las diferencias entre la vinculación dinámica y estática. Explique el uso de funciones virtuales y de la *vtable* en la vinculación dinámica.

**20.7** Plantee las diferencias entre herencia de interfaz y herencia de implementación. ¿Cómo es que las jerarquías de herencia diseñadas para la herencia de interfaz, difieren de aquellas diseñadas para la herencia de implementación?

**20.8** Plantee las diferencias entre las funciones virtuales y las funciones virtuales puras.

**20.9** (Verdadero/falso.) Todas las funciones virtuales de una clase base abstracta deben declararse como funciones virtuales puras.

**20.10** Sugiera uno o más niveles de clases base abstractas para la jerarquía **`Figura`** que explicamos en este capítulo (el primer nivel es **`Figura`** y el segundo nivel consiste en las clases **`FiguraBidimensional`** y **`FiguraTridimensional`**).

**20.11** ¿Cómo es que el polimorfismo promueve la extensibilidad?

**20.12** Se le ha pedido que desarrolle un simulador de vuelo que tendrá salidas gráficas elaboradas. Explique por qué la programación polimórfica sería especialmente efectiva para un problema de esta naturaleza.

**20.13** Desarrolle un paquete básico de gráficos. Utilice la clase **`Figura`** de la jerarquía de herencia del capítulo 19. Limítese a figuras bidimensionales como cuadrados, rectángulos, triángulos y círculos. Interactúe con el usuario. Permita que el usuario especifique la posición, el tamaño, la figura y los caracteres a utilizarse para dibujar cada figura. El usuario puede especificar muchos elementos de la misma figura. Conforme genere cada figura, coloque un apuntador **`Figura *`** a cada nuevo objeto de **`Figura`** en un arreglo. Cada clase tiene su propia función miembro dibujar. Escriba un administrador de pantalla polimórfico que recorra el arreglo (de preferencia utilizando un iterador), que envíe mensajes dibujar a cada objeto del arreglo para formar una imagen de pantalla. Vuelva a dibujar la imagen de la pantalla cada vez que el usuario especifique una figura adicional.

**20.14** En el ejercicio 20.12, usted desarrolló una jerarquía de clase **`Figura`** y definió las clases de la jerarquía. Modifique la jerarquía para que la clase **`Figura`** sea una clase base abstracta que contenga la interfaz de la jerarquía. Derive **`FiguraBidimensional`** y **`FiguraTridimensional`** de la clase **`Figura`**; estas clases también deben ser abstractas. Utilice una función **`virtual`** imprimir para desplegar el tipo y las dimensiones de cada clase. También incluya funciones **`area`** y **`volumen`** para que estos cálculos puedan realizarse para objetos de cada clase concreta en la jerarquía. Escriba un programa controlador que evalúe la jerarquía de clase **`Figura`**.

# 21

# Entrada/salida de flujo en C++

## Objetivos

- Comprender cómo utilizar la entrada/salida de flujo orientado a objetos en C++.
- Ser capaz de dar formato a entradas y salidas.
- Comprender la jerarquía de clases de entrada/salida de flujo.
- Comprender cómo introducir/desplegar objetos de tipos definidos por el usuario.
- Crear manipuladores de flujos definidos por el usuario.
- Determinar el éxito o el fracaso de operaciones de entrada/salida.
- Unir flujos de salida con flujos de entrada.




*La conciencia... no parece dividirse en pequeños bits... parece más natural describirla metafóricamente como un "río" o un "flujo".*
William James

*Todas las noticias que vale la pena escribir.*
Adolph S. Ochs

## Plan general

## 21.1 Introducción

Las bibliotecas estándar de C++ proporcionan un extenso conjunto de capacidades de entrada/salida. Este capítulo explica un rango suficiente de capacidades para realizar las operaciones más comunes de E/S, y da un vistazo general a las capacidades restantes. Algunas de las capacidades que presentamos aquí proporcionan una idea más completa de las capacidades de entrada/salida de C++.

Muchas de las características que describimos en este capítulo son orientadas a objetos. Al lector le parecerá interesante ver cómo se implementan dichas capacidades. Este estilo de E/S hace uso de otras características de C++, como referencias y sobrecarga de funciones y de operadores.

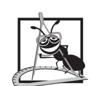
Como veremos, C++ utiliza *E/S con seguridad de tipos*. Cada operación de E/S se realiza automáticamente de manera sensible al tipo de dato. Si una función de E/S se definió adecuadamente para manejar un tipo de dato en particular, entonces se llama a esa función para manejar ese tipo de dato. Si no hay coincidencia entre el tipo de dato real y una función para manejar ese tipo de dato, se establece una indicación de error del compilador. Entonces, no pueden introducirse datos inapropiados al sistema (como puede ocurrir en C; una laguna en C que permite algunos errores extraños y sutiles).

Los usuarios pueden especificar la E/S de tipos definidos por el usuario, así como tipos estándar. Esta *extensibilidad* es una de las características más valiosas de C++.



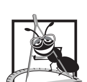
**Buena práctica de programación 21.1**

*En programas de C++ utilice exclusivamente la forma de E/S de C++, aunque el estilo de C para E/S esté disponible para los programadores en C++.*



**Observación de ingeniería de software 21.1**

*El estilo de E/S de C++ ofrece seguridad de tipos.*



**Observación de ingeniería de software 21.2**

*C++ ofrece un tratamiento común de E/S de tipos predefinidos y de tipos definidos por el usuario. Este tipo de tratamiento común facilita el desarrollo de software en general y la reutilización de software en particular.*

## 21.2 Flujos

La E/S en C++ ocurre por medio de *flujos* de bytes. Un flujo es simplemente una secuencia de bytes. En operaciones de entrada, los bytes fluyen desde un dispositivo (por ejemplo, un teclado, una unidad de disco, o una conexión de red) hacia la memoria principal. En operaciones de salida, los bytes fluyen desde la memoria principal hacia un dispositivo (por ejemplo, una pantalla, una impresora, una unidad de disco o una conexión de red).

La aplicación asocia su significado con los bytes. Los bytes pueden representar caracteres ASCII, datos en formato interno puro, imágenes gráficas, voz digital, video digital o cualquier otra clase de información que pueda necesitar una aplicación.

El trabajo de los mecanismos de E/S del sistema es mover los bytes desde dispositivos hacia la memoria y viceversa, de manera confiable. Con frecuencia, dichas transferencias involucran movimientos mecánicos, como la rotación de un disco o una cinta, o pulsar teclas en un teclado. El tiempo que se llevan estas transferencias normalmente es mucho, comparado con el tiempo que el procesador utiliza para manipular internamente los datos. Entonces, las operaciones de E/S requieren una planeación y un ajuste cuidadoso para garantizar el máximo rendimiento.

C++ proporciona capacidades de E/S de "bajo nivel" y de "alto nivel". Las capacidades de E/S de bajo nivel (es decir, *E/S sin formato*) generalmente especifican que un número de bytes deben simplemente transferirse desde un dispositivo hacia la memoria o desde la memoria hacia un dispositivo. En dichas transferencias, el byte individual es el elemento de interés. Tales capacidades de bajo nivel proporcionan transferencias de grandes volúmenes a alta velocidad, pero estas capacidades no son particularmente convenientes para la gente.

La gente prefiere una vista de alto nivel de la E/S (es decir, *E/S con formato*) en la que los bytes se agrupan en unidades significativas como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por el usuario. Estas capacidades orientadas a objetos son satisfactorias para la mayoría de las operaciones de E/S que no involucren el procesamiento de archivos de gran volumen.



**Tip de rendimiento 21.1**

*Utilice E/S sin formato, para un mejor rendimiento en el procesamiento de archivos de gran volumen.*

### 21.2.1 Archivos de encabezado de la biblioteca `iostream`

La biblioteca **`iostream`** de C++ proporciona cientos de capacidades de E/S. Diversos archivos de encabezado contienen partes de la interfaz de la biblioteca.

La mayoría de los programas en C++ incluyen el archivo de encabezado **`<iostream>`**, el cual declara servicios básicos necesarios para todas las operaciones de E/S de flujo. El archivo de encabezado **`<iostream>`** define los objetos **`cin`**, **`cout`**, **`cerr`** y **`clog`**, los cuales corresponden al flujo de entrada estándar, al flujo de salida estándar, flujo de error estándar sin búfer y flujo de error estándar con búfer, respectivamente. Estos servicios de E/S se proporcionan tanto sin formato como con formato.

El encabezado **`<iomanip>`** declara servicios útiles para realizar E/S con formato por medio de los *manipuladores parametrizados de flujo*.

Las implementaciones de C++ generalmente contienen otras bibliotecas relacionadas con la E/S, las cuales proporcionan capacidades específicas del sistema, como control de dispositivos de propósito especial para E/S de audio y video.

### 21.2.2 Clases y objetos para la entrada/salida de flujo

La biblioteca **`iostream`** contiene muchas clases para el manejo de una amplia variedad de operaciones de E/S. La clase **`istream`** soporta operaciones de entrada de flujo. La clase **`ostream`** soporta operaciones de salida de flujo. La clase **`iostream`** soporta las dos operaciones anteriores.

La clase **`istream`** y la clase **`ostream`** se derivan a través de la herencia simple de la clase base **`ios`**. La clase **`iostream`** se deriva a través de la herencia múltiple tanto de la clase **`istream`** como de la clase **`ostream`**. La figura 21.1 resume estas relaciones de herencia.

La sobrecarga de operadores proporciona una notación conveniente para realizar operaciones de entrada/salida. El operador de desplazamiento a la izquierda (**`<<`**) se sobrecarga para designar la salida de flujo, y se le conoce como *operador de inserción de flujo*. El operador de desplazamiento a la derecha (**`>>`**) se sobrecarga para designar la entrada de flujo, y se le conoce como *operador de extracción de flujo*. Estos operadores se utilizan con los objetos de flujo estándar **`cin`**, **`cout`**, **`cerr`** y **`clog`**, y comúnmente con los objetos de flujo definidos por el usuario.

El objeto predefinido **`cin`** es una instancia de la clase istream, y se dice que está “unido con” (o conectado con) el dispositivo de entrada estándar, que generalmente es el teclado. El operador de extracción de flujo (**`>>`**), como se utiliza en la siguiente instrucción, ocasiona que se introduzca un valor para la variable entera **`calificacion`** (suponiendo que **`calificacion`** se declaró como una variable **`int`**) desde **`cin`** hacia la memoria:

```
cin >> calificacion;    // los datos “fluyen” en la dirección de las flechas
                        // hacia la derecha
```

Observe que la operación de extracción de flujo es “lo suficientemente inteligente” para “saber” de qué tipo de dato se trata. Si suponemos que **`calificacion`** se declaró adecuadamente, no es necesario especificar información adicional sobre el tipo para utilizarla con el operador de extracción de flujo (como es el caso, incidentalmente, en el estilo de E/S de C).

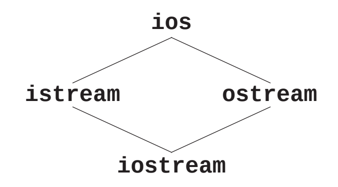



**Figura 21.1** Parte de la jerarquía de la clase de E/S de flujo.

El objeto predefinido **cout** es una instancia de la clase **ostream**, y se dice que está "unido con" el dispositivo de salida estándar, que generalmente es la pantalla. El operador de inserción de flujo (**<<**), como se utiliza en la siguiente instrucción, ocasiona que se despliegue el valor de la variable entera **calificacion** (suponiendo que **calificacion** se declaró como una variable **int**) desde la memoria hacia el dispositivo de salida estándar:

```
cout << calificacion;   // los datos "fluyen" en la dirección de las flechas
                        // hacia la izquierda
```

Observe que la operación de inserción de flujo es "lo suficientemente inteligente" para "saber" el tipo de **calificacion** (suponiendo que **calificacion** se declaró adecuadamente), por lo que no es necesario especificar información adicional sobre su tipo para utilizarla con el operador de inserción de flujo.

El objeto predefinido **cerr** es una instancia de la clase **ostream**, y se dice que está "unido con" el dispositivo de error estándar. Las salidas del objeto **cerr** son sin búfer, lo cual significa que cada inserción de flujo hacia **cerr** ocasiona que su salida aparezca inmediatamente; esto es adecuado para notificar con rapidez al usuario sobre los errores.

El objeto predefinido **clog** es una instancia de la clase **ostream**, y también se dice que está "unido con" el dispositivo de error estándar. Las salidas del objeto **clog** son con búfer, lo cual significa que cada inserción de flujo hacia **clog** podría ocasionar que su salida se mantuviera en el búfer hasta que éste se llene, o hasta que se vacíe.

El procesamiento de archivos en C++ utiliza las clases **ifstream** para realizar operaciones de entrada de archivos, **ofstream** para operaciones de salida de archivos y **fstream** para operaciones de entrada/salida de archivos. La clase **ifstream** hereda desde la clase **istream**, la clase **ofstream** hereda desde la clase **ostream**, y la clase **fstream** hereda desde la clase **iostream**. La figura 21.2 resume las diversas relaciones de herencia entre las clases de entrada/salida. Existen muchas más clases en la jerarquía completa de la clase de E/S de flujo que soportan la mayoría de las instalaciones, pero las clases que mostramos aquí proporcionan casi todas las capacidades que los programadores necesitarán. Para mayor información, vea la referencia de la biblioteca de clases para su sistema C++ relacionada con el procesamiento de archivos.

## 21.3 Salida de flujo

La clase **ostream** de C++ proporciona la habilidad de realizar operaciones de salida con formato y sin formato. Las capacidades de salida incluyen la salida de tipos de datos estándar con el operador de inserción de flujo; la salida de caracteres con la función miembro **put**; la salida sin formato con la función miembro **write** (sección 21.5); la salida de enteros en formato decimal, octal y hexadecimal (sección 21.6.1); la salida de valores de punto flotante con diversas precisiones (sección 21.6.2), con puntos decimales forzados (sección 21.7.2), en notación científica y en notación fija (sección 21.7.6); la salida de datos justificados en campos con anchos no asignados (sección 21.7.3); la salida de datos en campos rellenos con caracteres especificados (sección 21.7.4); y la salida de letras mayúsculas en notación científica y hexadecimal (sección 21.7.7).

### 21.3.1 Operador de inserción de flujo

La salida de flujo puede realizarse con el operador de inserción de flujo (es decir, con el operador << sobrecargado). El operador << se sobrecarga para desplegar elementos de datos de tipos integrados, para desplegar

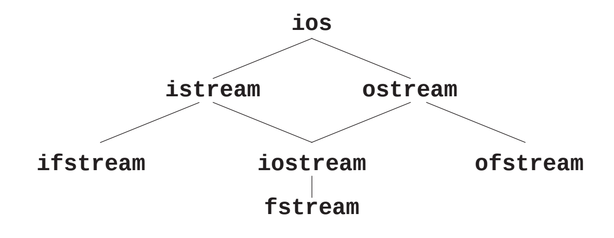



**Figura 21.2** Parte de la jerarquía de la clase de E/S de flujo con las principales clases de procesamiento de archivos.

cadenas y para desplegar valores de apuntadores. La sección 21.9 muestra cómo sobrecargar el operador << para desplegar elementos de datos de tipos definidos por el usuario. La figura 21.3 muestra la salida de una cadena por medio de una sola instrucción de inserción de flujo. Es posible utilizar múltiples instrucciones de inserción, como en la figura 21.4. Cuando se ejecuta este programa, produce la misma salida que el programa de la figura 21.3.

El efecto de la secuencia de escape **\n** (nueva línea) también se logra con el *manipulador de flujo* **endl** (fin de línea), como en la figura 21.5. El manipulador de flujo **endl** despliega un carácter de nueva línea y, además, vacía el búfer de salida (es decir, ocasiona que el búfer de salida se despliegue inmediatamente, incluso si no está lleno). El búfer de salida también puede vaciarse con

```
cout << flush;
```

en la sección 21.6 explicamos con detalle los manipuladores de flujo.

Las expresiones pueden desplegarse como muestra la figura 21.6.



### Buena práctica de programación 21.2

*Cuando despliegue expresiones, colóquelas entre paréntesis para evitar problemas con la precedencia de los operadores de la expresión y el operador <<.*

```
// Figura 21.3: fig21_03.cpp
// Despliega una cadena mediante la inserción de flujo
#include <iostream>

using std::cout;

int main()
{
   cout << “Bienvenido a C++!\n”;

   return 0;
} // fin de la función main
```

```
Bienvenido a C++!
```

**Figura 21.3** Despliegue de una cadena por medio de la inserción de flujo.

```
// Figura 21.4: fig21_04.cpp
// Despliega una cadena mediante el uso de dos inserciones de flujo
#include <iostream>

using std::cout;

int main()
{
   cout << “Bienvenido a “;
   cout << “C++!\n”;

   return 0;
} // fin de la función main
```

```
Bienvenido a C++!
```

**Figura 21.4** Despliegue de una cadena por medio de dos inserciones de flujo.

```
// Figura 21.5: fig21_05.cpp
// Uso del manipulador de flujo endl
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << “Bienvenido a “;
   cout << “C++!”;
   cout << endl;      // fin del manipulador de flujo

   return 0;
} // fin de la función end main
```

```
Bienvenido a C++!
```

**Figura 21.5** Uso del manipulador de flujo **endl**.

```
// Figura 21.6: fig21_06.cpp
// Despliega los valores de una expresión.
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << “47 + 53 es “;

   // no se requieren los paréntesis; se utilizan para mayor claridad
  cout << ( 47 + 53 );      // expresión
   cout << endl;

   return 0;
} // fin de la función main
```

```
47 + 53 es 100
```

**Figura 21.6** Despliegue de los valores de una expresión.

## 21.3.2 Operadores para la inserción/extracción de flujo en cascada

Los operadores << y >> pueden utilizarse *en cascada*, como muestra la figura 21.7.

Las diversas inserciones de flujo de la figura 21.7 se ejecutan como si se hubieran escrito

```
( ( ( cout << “47 mas 53 es ” ) << ( 47 + 53 ) ) << endl );
```

(es decir, << asocia de izquierda a derecha). Esta manera de colocar en cascada los operadores de inserción de flujo está permitida debido a que los operadores << sobrecargados devuelven una referencia hacia el objeto de su operando izquierdo (es decir, **cout**). Por lo tanto, la expresión entre paréntesis que se encuentra más hacia la izquierda

```
( cout << “47 mas 53 es ” )
```

```
// Figura 21.7: fig21_07.cpp
// operador sobrecargado << en cascada
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << “47 mas 53 es “ << ( 47 + 53 ) << endl;

   return 0;
} // fin de la función main
```

```
47 + 53 es 100
```

**Figura 21.7** Operador sobrecargado << en cascada.

despliega la cadena de caracteres especificada, y devuelve una referencia a **`cout`**. Esto permite que la expresión entre paréntesis que se encuentra en medio se evalúe como

**`( cout << ( 47 + 53 ) )`**

la cual despliega el valor entero 100, y devuelve una referencia a **`cout`**. Entonces, la expresión entre paréntesis que se encuentra más a la derecha se evalúa como

**`cout << endl`**

la cual despliega una nueva línea, vacía **`cout`** y devuelve una referencia a **`cout`**. Esta última devolución no se utiliza.

### 21.3.3 Salida de variables `char *`

En la E/S al estilo de C, es necesario que el programador proporcione información sobre el tipo. C++ determina automáticamente los tipos de los datos; una buena mejora al lenguaje C. Sin embargo, algunas veces esto es “un estorbo”. Por ejemplo, sabemos que una cadena de caracteres es de tipo **`char *`**. Suponga que queremos imprimir el valor de ese apuntador, es decir, la dirección en memoria del primer carácter de esa cadena. Pero, el operador << se sobrecargó para que imprimiera datos de tipo **`char *`** como una cadena terminada con null. La solución es convertir el tipo del apuntador a **`void *`** (esto debe hacerse para cualquier apuntador que el programador quiera desplegar como una dirección). La figura 21.8 muestra la impresión de una variable **`char *`** en los formatos de cadena y de dirección. Observe que la dirección se imprime como un número hexadecimal (base 16). En las secciones 21.6.1, 21.7.4, 21.7.5 y 21.7.7 hablamos más sobre el control de las bases de los números. [*Nota:* La salida del programa correspondiente a la figura 21.8 puede diferir de compilador a compilador.]

```
// Figura 21.8: fig21_08.cpp
// Impresión de la dirección almacenada en una variable char*
#include <iostream>

using std::cout;
using std::endl;

int main()
{
```

**Figura 21.8** Impresión de la dirección almacenada en una variable **`char *`**. (Parte 1 de 2.)

```
   char *cadena = "prueba";

   cout << "El valor de la cadena es: " << cadena
        << "\nEl valor de static_cast< void * >( cadena ) es: "
        << static_cast< void * >( cadena ) << endl;
   return 0;
} // fin de la función main
```

```
El valor de la cadena es: prueba
El valor de static_cast< void * >( cadena ) es: 0046C073
```

**Figura 21.8** Impresión de la dirección almacenada en una variable **`char *`**. (Parte 2 de 2.)

### 21.3.4 Salida de caracteres por medio de la función miembro `put`; funciones `put` en cascada

La función miembro **`put`** despliega un carácter como en

**`cout.put( 'A' ),`**

la cual despliega en la pantalla una A. Las llamadas a **`put`** pueden hacerse en cascada como en

**`cout.put( 'A' ).put( '\n' );`**

lo cual despliega la letra A, seguida por un carácter de nueva línea. Como sucede con <<, la instrucción anterior se ejecuta de esta manera, debido a que el operador punto (**`.`**) asocia de izquierda a derecha, y la función miembro **`put`** devuelve una referencia al objeto **`ostream`** que recibió el mensaje **`put`** (una llamada a la función). La función **`put`** también puede invocarse con una expresión con valores ASCII, como en **`cout.put( 65 )`**, la cual también despliega una **`A`**.

## 21.4 Entrada de flujo

Ahora consideremos la entrada de flujo. Ésta puede realizarse con el operador de extracción de flujo (es decir, el operador >> sobrecargado). Este operador normalmente ignora los *caracteres blancos* (como espacios, tabuladores y nuevas líneas) en el flujo de entrada. Más adelante veremos cómo cambiar este comportamiento. El operador de extracción de flujo devuelve cero (falso) cuando se encuentra un fin de archivo en un flujo; de lo contrario, devuelve una referencia hacia el objeto que recibió el mensaje de extracción (por ejemplo, **`cin`** en la expresión **`cin`** >> **`calificacion`**). Cada flujo contiene un conjunto de *bits de estado* que se utiliza para controlar el estado del flujo (es decir, el formato, la asignación de errores de estado, etcétera). La extracción de flujo ocasiona que, si se introducen datos de tipo incorrecto, se establezca el **`failbit`** del flujo, y ocasiona que, si la operación falla, se establezca el **`badbit`** del flujo. Pronto veremos cómo evaluar estos bits después de una operación de E/S. Las secciones 21.7 y 21.8 explican con detalle los bits de estado de un flujo.

### 21.4.1 Operador de extracción de flujo

Para leer dos enteros, utilice el objeto **`cin`** y el operador de extracción de flujo >> sobrecargado, como en la figura 21.9. Observe que las operaciones de extracción de flujo también pueden realizarse en cascada.

La relativa alta precedencia de los operadores >> y << puede ocasionar problemas. Por ejemplo, el programa de la figura 21.10 no se compilará apropiadamente sin los paréntesis alrededor de la expresión condicional. El lector debe verificar esto.



**Error común de programación 21.1**

*Intentar realizar una lectura desde un **`ostream`** (o desde cualquier otro flujo de sólo salida), es un error.*



**Error común de programación 21.2**

*Intentar escribir en un **`istream`** (o en cualquier otro flujo de sólo entrada), es un error.*

```
// Figura 21.9: fig21_09.cpp
// Calcula la suma de dos enteros introducidos desde el teclado
// con cin y el operador de extracción de flujo.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x, y;

   cout << “Introduzca dos enteros: “;
   cin >> x >> y;
   cout << “La suma de “ << x << “ y “ << y << “ es: “
        << ( x + y ) << endl;

   return 0;
} // fin de la función main
```

```
Introduzca dos enteros: 30 92
La suma de 30 y 92 es: 122
```

**Figura 21.9** Cálculo de la suma de dos enteros introducidos desde el teclado con **cin** y el operador de extracción de flujo.

```
// Figura 21.10: fig21_10.cpp
// Evita un problema de precedencia entre el operador de inserción
// de flujo y el operador condicional.
// Se requieren parántesis alrededor de la expresión condicional.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x, y;

   cout << “Introduzca dos enteros: “;
   cin >> x >> y;
   cout << x << ( x == y ? “ es” : “ no es” )
        << “ igual a “ << y << endl;

   return 0;
} // fin de la función main
```

```
Introduzca dos enteros: 7 5
7 no es igual a 5
```

```
Introduzca dos enteros: 8 8
8 es igual a 8
```

**Figura 21.10** Cómo evitar el problema de precedencia entre el operador de inserción de flujo y el operador condicional.



### Error común de programación 21.3

*No proporcionar paréntesis para forzar la precedencia adecuada, cuando se utiliza la relativa alta precedencia del operador de inserción de flujo << o del operador de extracción de flujo >>, es un error.*

Una forma popular para introducir una serie de valores es por medio de la operación de extracción de flujo en la condición de continuación de ciclo correspondiente a un ciclo **`while`**, La extracción devuelve **`falso`** (0), cuando se encuentra el fin de archivo. Considere el programa de la figura 21.11, el cual localiza la calificación más alta de un examen. Suponga que el número de calificaciones no se conoce por adelantado, y que el usuario escribirá el fin de archivo para indicar que se introdujeron todas las calificaciones. La condición **`while`**, ( **`cin`** >> **`calificacion`** ), se vuelve 0 (la cual se interpreta como **`falso`**) cuando el usuario introduce el fin de archivo.



### Tip de portabilidad 21.1

*Cuando indique al usuario cómo terminar la introducción de datos desde el teclado, solicítele que "introduzca el fin de archivo para finalizar la entrada de datos", en lugar de solicitarle un* <ctrl>d *(UNIX y Macintosh) o* <ctrl>z *(PC y VAX).*

En la figura 21.11, **`cin`** >> **`calificacion`** puede utilizarse como una condición, ya que la clase base **`ios`** (de la que hereda **`istream`**) proporciona un operador sobrecargado de conversión de tipo, el cual con-

```
// Figura 21.11: fig21_11.cpp
// Operador de extracción de flujo que devuelve falso o fin-de-archivo.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int calificacion, califMasAlta = -1;

   cout << "Introduzca la calificacion (introduzca fin de archivo para
   terminar): ";
   while ( cin >> calificacion ) {
      if ( calificacion > califMasAlta )
         califMasAlta = calificacion;

      cout << "Introduzca la calificacion (introduzca fin de archivo para
      terminar): ";
   } // fin de while

   cout << "\n\nLa calificacion mas alta es: " << califMasAlta << endl;
   return 0;
} // fin de la función main
```

```
Introduzca la calificacion (introduzca fin de archivo para terminar): 67
Introduzca la calificacion (introduzca fin de archivo para terminar): 87
Introduzca la calificacion (introduzca fin de archivo para terminar): 73
Introduzca la calificacion (introduzca fin de archivo para terminar): 95
Introduzca la calificacion (introduzca fin de archivo para terminar): 34
Introduzca la calificacion (introduzca fin de archivo para terminar): 99
Introduzca la calificacion (introduzca fin de archivo para terminar): ^Z
La calificacion mas alta es: 99
```

**Figura 21.11** Operador de extracción de flujo que devuelve falso en un fin de archivo.

vierte un flujo en un apuntador de tipo **void** *. El valor del apuntador devuelto es 0 (falso), si ocurrió un error mientras se intentaba leer un valor o si se encontró el indicador de fin de archivo. El compilador puede utilizar implícitamente el operador de conversión de tipo **void** *.

### 21.4.2 Funciones miembro get y getline

La función **get** sin argumento alguno introduce un carácter desde el flujo designado (incluso si se trata de un carácter blanco), y devuelve este carácter como el valor de la llamada a la función. Esta versión de **get** devuelve un **EOF** cuando se encuentra el fin de archivo en el flujo.

La figura 21.12 muestra el uso de funciones miembro **eof** y **get** en un flujo de entrada **cin**, y el uso de la función miembro **put** en un flujo de salida **cout**. El programa primero imprime el valor de **cin.eof()** [es decir, falso (**0** en la salida)], para mostrar que no se ha encontrado el fin de archivo en **cin.** El usuario introduce una línea de texto y oprime *Entrar*, seguida por el indicador de fin de archivo (*<ctrl>z* en sistemas compatibles con la PC de IBM, *<ctrl>d* en sistemas UNIX y Macintosh). El programa lee cada carácter y lo saca hacia **cout**, utilizando la función miembro **put**. Cuando se encuentra el fin de archivo, el **while** termina, y **cin.eof()** (ahora verdadero) se imprime de nuevo (**1** en la salida) para mostrar que el fin de archivo se estableció en **cin**. Observe que este programa utiliza la versión **istream** de la función miembro **get** que no toma argumentos, y que devuelve el carácter que se introduce.

La función miembro **get** con una referencia de carácter como argumento introduce el siguiente carácter desde el flujo de entrada (incluso si es un carácter blanco), y lo almacena en el argumento de carácter. Esta ver-

```
// Figura 21.12: fig21_12.cpp
// Uso de las funciones miembro get, put y eof.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   char c;

   cout << "Antes de la entrada, cin.eof() es " << cin.eof()
        << "\nIntroduzca una frase seguida por fin-de-archivo:\n";

   while ( ( c = cin.get() ) != EOF )
      cout.put( c );

   cout << "\nEn este sistema, EOF es: " << c;
   cout << "\nDespues de la salida, cin.eof() es " << cin.eof() << endl;
   return 0;
} // fin de la función main
```

```
Antes de la entrada, cin.eof() es 0
Introduzca una frase seguida por fin-de-archivo:
Probando las funciones miembro get y put
Probando las funciones miembro get y put
^Z

En este sistema, EOF es:
Despues de la salida, cin.eof() es 1
```

**Figura 21.12** Uso de las funciones miembro **get**, **put** y **eof**.

sión de **get** devuelve 0 cuando se encuentra el fin de archivo; de lo contrario devuelve una referencia hacia el objeto **istream** para el que se invocó a la función miembro **get**.

Una tercera versión de la función miembro **get** toma tres argumentos: un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor predeterminado **‘\n’**). Esta versión lee caracteres desde el flujo de entrada; lee un carácter menos que el número máximo especificado de caracteres y termina, o finaliza tan pronto como lee el delimitador. Se inserta un carácter nulo para terminar la cadena de entrada en el arreglo de caracteres que el programa utiliza como búfer. El delimitador no se coloca en el arreglo de caracteres, pero permanece en el flujo de entrada (el delimitador será el siguiente carácter que se lea). Entonces, el resultado de un segundo **get** consecutivo es una línea vacía, a menos que el carácter delimitador se elimine del flujo de entrada. La figura 21.13 compara la introducción de datos por medio de **cin** con extracción de flujo (lo cual lee caracteres hasta que se encuentra un carácter blanco) con la introducción de datos por medio de **cin.get**. Observe que la llamada a **cin.get** no especifica un carácter delimitador, por lo que se utiliza **‘\n’** como predeterminado.

La función miembro **getline** funciona como la tercera versión de la función miembro **get**, e inserta un carácter nulo después de la línea en el arreglo de caracteres. La función **getline** elimina del flujo al delimitador (es decir, lee el carácter y lo descarta), pero no lo almacena en el arreglo de caracteres. El programa de la figura 21.14 muestra el uso de la función miembro **getline** para introducir una línea de texto.

```
// Figura 21.13: fig21_13.cpp
// Compara la entrada de una cadena con cin y cin.get.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   const int TAMANIO = 80;
   char bufer1[ TAMANIO ], bufer2[ TAMANIO ];

   cout << “Introduzca una frase:\n”;
   cin >> bufer1;
   cout << “\nLa cadena leida con cin fue:\n”
        << bufer1 << “\n\n”;

   cin.get( bufer2, TAMANIO );
   cout << “La cadena leida con cin.get fue:\n”
      << bufer2 << endl;

   return 0;
} // fin de la función main
```

```
Introduzca una frase:
Compara la introuccion de cadenas mediante cin y cin.get

La cadena leida con cin fue:
Compara

La cadena leida con cin.get fue:
 la introuccion de cadenas mediante cin y cin.get
```

**Figura 21.13** Comparación de la entrada de una cadena por medio de **cin** con extracción de flujo, con la entrada de una cadena por medio de **cin.get**.

```
// Figura 21.14: fig21_14.cpp
// Entrada de caracteres con la función miembro getline.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   const TAMANIO = 80;
   char bufer[ TAMANIO ];

   cout << "Introduzca una frase:\n";
   cin.getline( bufer, TAMANIO );

   cout << "\nLa frase introducida es:\n" << bufer << endl;
   return 0;
} // fin de la función main
```

```
Introduzca una frase:
Uso de la funcion miembro getline

La frase introducida es:
Uso de la funcion miembro getline
```

**Figura 21.14** Entrada de un carácter por medio de la función miembro **getline**.

### 21.4.3 Funciones miembro de istream: peek, putback e ignore

La función miembro **ignore** pasa por alto un número designado de caracteres (el predeterminado es un carácter), o termina hasta que encuentra el delimitador designado (el delimitador predeterminado es **EOF**, el cual ocasiona que **ignore** salte hacia el fin del archivo cuando realiza una lectura desde un archivo).

La función miembro **putback** coloca el carácter obtenido previamente por un **get** del flujo de entrada, de regreso hacia ese flujo. Esta función es útil para aplicaciones que exploran un flujo de entrada en busca de un campo que comience con un carácter en especial. Cuando ese carácter se introduce, la aplicación coloca el carácter de nuevo en el flujo, para que éste pueda incluirse en los datos de entrada.

La función miembro **peek** devuelve el siguiente carácter de un flujo de entrada, pero no lo elimina del flujo.

### 21.4.4 E/S con seguridad de tipos

C++ ofrece la *E/S con seguridad de tipos*. Los operadores << y >> se sobrecargan para aceptar elementos de datos de tipos específicos. Si se procesan datos inesperados, se establecen varias banderas de error que el usuario puede evaluar para determinar si una operación de E/S se llevó a cabo con éxito, o si falló. De esta manera, el programa "permanece bajo control". En la sección 21.8 explicaremos estas banderas de error.

## 21.5 E/S sin formato por medio de read, gcount y write

La *entrada/salida sin formato* se realiza por medio de las funciones miembro **read** y **write**. Cada una de ellas introduce o despliega cierto número de bytes hacia o desde un arreglo de caracteres en memoria. Estos bytes no tienen formato alguno, simplemente se introducen o se despliegan como bytes puros. Por ejemplo, la llamada

```
char bufer[] = "FELIZ CUMPLEANIOS";
cout.write( bufer, 10 );
```

despliega los primeros 10 bytes de **bufer** (incluso los caracteres nulos que ocasionarían que la salida con **cout** y << terminara). Debido a que una cadena de caracteres da como resultado la dirección de su primer carácter, la llamada

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

despliega los 10 primeros caracteres del alfabeto.

La función miembro **read** introduce un número designado de caracteres en un arreglo de caracteres. Si se leen menos caracteres que el número designado, se establece **failbit**. Pronto veremos cómo determinar si se estableció un **failbit** (vea la sección 21.8). La función miembro **gcount** reporta el número de caracteres leídos por la última operación de entrada.

La figura 21.15 muestra las funciones miembro de **istream**, **read** y **gcount**, y la función miembro de **ostream**, **write**. El programa introduce, por medio de **read**, 20 caracteres (a partir de una secuencia de entrada más grande) en el arreglo de caracteres **bufer**; determina, por medio de **gcount**, el número de caracteres introducidos; y despliega, por medio de **write**, los caracteres de **bufer**.

## 21.6 Manipuladores de flujo

C++ proporciona varios *manipuladores de flujo* que realizan tareas de formato. Los manipuladores de flujo proporcionan capacidades como establecer el ancho de un campo, establecer precisiones, establecer y restablecer banderas de formato, establecer el carácter de llenado en un campo, vaciado de flujos, insertar una nueva línea en el flujo de salida y vaciar el flujo, insertar un carácter nulo en el flujo de salida e ignorar espacios blancos en el flujo de entrada. En las siguientes secciones, describiremos estas características.

### 21.6.1 Base de un flujo de enteros: dec, oct, hex y setbase

Los enteros normalmente se interpretan como valores decimales (base 10). Para cambiar la base en la que se interpretan los enteros de un flujo, inserte el manipulador **hex** para establecer la base en hexadecimal (base 16),

```
// Figura 21.15: fig21_15.cpp
// E/S sin formato con read, gcount y write.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   const int TAMANIO = 80;
   char bufer[ TAMANIO ];

   cout << "Introduzca una frase:\n";
   cin.read( bufer, 20 );
   cout << "\nLa frase introducida fue:\n";
   cout.write( bufer, cin.gcount() );
   cout << endl;
   return 0;
} // fin de la función main
```

```
Introduzca una frase:
Uso de las funciones miembro read, write y gcount

La frase introducida fue:
Uso de las funciones
```

**Figura 21.15** E/S sin formato con **read**, **gcount** y **write**.

o inserte el manipulador **oct** para establecer la base en octal (base 8). Inserte el manipulador de flujo **dec** para restablecer la base del flujo en decimal.

La base de un flujo también puede cambiarse por medio del manipulador de flujo **setbase**, el cual toma un argumento entero de **10**, **8** o **16** para establecer la base. El manipulador de flujo **setbase** toma un argumento, por lo que se le conoce como *manipulador parametrizado de flujo*. Para utilizar **setbase** o cualquier otro manipulador parametrizado es necesario incluir el archivo de encabezado **<iomanip>**. La base del flujo permanece igual, hasta que explícitamente se modifique. La figura 21.16 muestra el uso de los manipuladores de flujo **hex**, **oct**, **dec** y **setbase**.

### 21.6.2 Precisión de punto flotante (precision, setprecision)

Es posible controlar la *precisión* de números de punto flotante (es decir, el número de dígitos a la derecha del punto decimal), por medio del manipulador de flujo **setprecision** o por medio de la función miembro **precision**. Una llamada a cualquiera de éstas ocasiona que se establezca la precisión para todas las operaciones de salida subsiguientes, hasta la siguiente llamada para establecer la precisión. La función miembro

```
// Figura 21.16: fig21_16.cpp
// Uso de los manipuladores de flujo hex, oct, dec y setbase.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::hex;
using std::dec;
using std::oct;
using std::setbase;

int main()
{
   int n;

   cout << “Introduzca un numero decimal: “;
   cin >> n;

   cout << n << “ en hexadecimal es: “
        << hex << n << ‘\n’
        << dec << n << “ en octal es: “
        << oct << n << ‘\n’
        << setbase( 10 ) << n << “ en decimal es: “
        << n << endl;

   return 0;
} // fin de la función main
```

```
Introduzca un numero decimal: 20
20 en hexadecimal es: 14
20 en octal es: 24
20 en decimal es: 20
```

**Figura 21.16** Uso de los manipuladores de flujo **hex**, **oct**, **dec** y **setbase**.

**precision** sin argumentos devuelve la precisión actual establecida. El programa de la figura 21.17 utiliza tanto la función miembro **precision** como el manipulador **setprecision**, para imprimir una tabla que muestra la raíz cuadrada de **2**, con precisiones que varían de **0** a **9**.

## 21.6.3 Ancho de campo (setw, width)

La función miembro de **ios**, **width**, establece el ancho de un campo (es decir, el número de posiciones de carácter en las que debe desplegarse un valor, o el número de caracteres que debe introducirse) y devuelve el ancho anterior. Si los valores procesados son menos que el ancho del campo, se insertan *caracteres de relleno*. Un valor más amplio que el ancho designado no se truncará; se imprimirá el número completo.



**Error común de programación 21.4**

*Un ancho establecido aplica sólo para la siguiente inserción o extracción; después de eso, el ancho se establece implícitamente en **0** (es decir, los valores desplegados simplemente serán tan amplios como sea necesario). La función **width** sin argumentos devuelve el valor establecido actual. Asumir que el ancho establecido se aplica a todas las salidas subsiguientes, es un error lógico.*

```
// Figura 21.17: fig21_17.cpp
// Control de la precisión de valores de punto flotante
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include <cmath>

int main()
{
   double raiz2 = sqrt( 2.0 );
   int posiciones;

   cout << setiosflags( ios::fixed)
        << “Raiz cuadrada de 2 con precisiones 0-9.\n”
        << “Precision establecida por la “
        << “funcion miembro precision:” << endl;

   for ( posiciones = 0; posiciones <= 9; posiciones++ ) {
      cout.precision( posiciones );
      cout << raiz2 << ‘\n’;
   } // fin de for

   cout << “\nPrecision establecida por el “
        << “manipulador setprecision:\n”;

   for ( posiciones = 0; posiciones <= 9; posiciones++ )
      cout << setprecision( posiciones ) << raiz2 << ‘\n’;

```

**Figura 21.17** Control de la precisión de valores de punto flotante. (Parte 1 de 2.)

```
   return 0;
} // fin de la función main
```

```
Raiz cuadrada de 2 con precisiones 0-9.
Precision establecida por la funcion miembro precision:
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

```
Precision establecida por el manipulador setprecision:
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

**Figura 21.17** Control de la precisión de valores de punto flotante. (Parte 2 de 2.)



**Error común de programación 21.5**

*Cuando no proporciona un ancho de campo suficiente para manejar las salidas, éstas se imprimen tan amplias como sea necesario, lo que probablemente ocasione dificultades para leerlas.*

La figura 21.18 muestra el uso de la función miembro **`width`** tanto en la entrada como en la salida. Observe que al introducir valores en un arreglo **`char`**, se leerá un número máximo de caracteres de uno menos que el ancho, ya que se prevé que el carácter nulo se colocará en la cadena de entrada. Recuerde que la extracción de flujo termina cuando se encuentra un carácter blanco. El manipulador de flujo **`setw`** también puede utilizarse para establecer el ancho del campo. [*Nota:* Cuando se le indica al usuario que realice una entrada, éste debe introducir una línea de texto y oprimir *Entrar*, seguida por el indicador de fin de archivo (*<ctrl>z* en sistemas compatibles con la PC de IBM, o *<ctrl>d* en sistemas UNIX y Macintosh).] Observe que cuando se introduce cualquier otra cosa que no sea un arreglo **`char`**, **`width`** y **`setw`** se ignoran.

```
// fig21_18.cpp
// Demuestra la función miembro width
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
```

**Figura 21.18** Demostración de la función miembro **`width`**. (Parte 1 de 2.)

```
   int w = 4;
   char cadena[ 10 ];

   cout << "Introduzca una frase:\n";
   cin.width( 5 );

   while ( cin >> cadena ) {
      cout.width( w++ );
      cout << cadena << endl;
      cin.width( 5 );
   } // fin de while

   return 0;
} // fin de la función main
```

```
Introduzca una frase:
Esta es una prueba de la funcion miembro width
Esta
   es
   una
   prue
      ba
       de
        la
       func
         ion
         miem
           bro
           widt
               h
```

**Figura 21.18** Demostración de la función miembro **`width`**. (Parte 2 de 2.)

## 21.6.4 Manipuladores definidos por el usuario

Los usuarios pueden crear sus propios manipuladores de flujo. La figura 21.19 muestra la creación y el uso de los nuevos manipuladores de flujo **`campana`**, **`retorno`** (retorno de carro), **`tab`** y **`finDeLinea`**. Los usuarios también pueden crear sus propios manipuladores parametrizados de flujo; consulte el manual de su equipo para obtener las instrucciones sobre cómo hacer esto.

```
// Figura 21.19: fig21_19.cpp
// Creación y prueba de manipuladores de flujo sin parámetros
// definidos por el usuario.
#include <iostream>

using std::ostream;
using std::cout;
using std::flush;

// manipuladores de campana (mediante el uso de la secuencia de escape \a)
ostream& campana( ostream& salida ) { return salida << '\a'; }

```

**Figura 21.19** Creación y prueba de manipuladores de flujo sin parámetros definidos por el usuario. (Parte 1 de 2.)

```
// manipulador de retorno (mediante el uso de la secuencia de escape \r)
ostream& ret( ostream& salida ) { return salida << ‘\r’; }

// manipulador tab (mediante el uso de la secuencia de escape \t)
ostream& tab( ostream& salida ) { return salida << ‘\t’; }

// manipulador finLinea (mediante el uso de la secuencia de escape \n
// y la función miembro flush)
ostream& finLinea( ostream& salida )
{
   return salida << ‘\n’ << flush;
} // fin de la función fin línea

int main()
{
   cout << “Prueba del manipulador tab:” << finLinea
        << ‘a’ << tab << ‘b’ << tab << ‘c’ << finLinea
        << “Prueba de los manipuladores ret y campana:”
        << finLinea << “..........”;
   cout << campana;
   cout << ret << “-----” << finLinea;
   return 0;
} // fin de la función main
```

```
Prueba del manipulador tab:
a       b       c
Prueba de los manipuladores ret y campana:
-----.....
```

**Figura 21.19** Creación y prueba de manipuladores de flujo sin parámetros definidos por el usuario. (Parte 2 de 2.)

## 21.7 Estados de formato de flujo

Diversas *banderas de formato* especifican los tipos de formato a realizarse durante operaciones de E/S de flujo. Las funciones miembro **`setf`**, **`unsetf`**, y **`flags`** controlan la configuración de las banderas.

### 21.7.1 Banderas de estado de formato

Cada una de las banderas de estado de formato que aparece en la figura 21.20 (y algunas otras que no aparecen) se definen como una enumeración en la clase **`ios`**, y las explicaremos en las siguientes secciones.

| Bandera de estado de formato | Descripción |
|---|---|
| **`ios::skipws`** | Ignora los caracteres blancos de un flujo de entrada. |
| **`ios::left`** | Justifica a la izquierda la salida en un campo. Si es necesario, aparecen caracteres de relleno a la derecha. |
| **`ios::right`** | Justifica a la derecha la salida en un campo. Si es necesario, aparecen caracteres de relleno a la izquierda. |

**Figura 21.20** Banderas de estado de formato. (Parte 1 de 2.)

| Bandera de estado de formato | Descripción |
|---|---|
| **ios::internal** | Indica que el signo de un número debe justificarse a la izquierda en un campo, y que la magnitud del número debe justificarse a la derecha en ese mismo campo (es decir, aparecen caracteres de relleno entre el signo y el número). |
| **ios::dec** | Especifica que los enteros deben tratarse como valores decimales (base 10). |
| **ios::oct** | Especifica que los enteros deben tratarse como valores octales (base 8). |
| **ios::hex** | Especifica que los enteros deben tratarse como valores hexadecimales (base 16). |
| **ios::showbase** | Especifica que la base de un número debe imprimirse adelante de éste (un cero a la izquierda para los octales; un **0x** o **0X** a la izquierda para los hexadecimales). |
| **ios::showpoint** | Especifica que los números de punto flotante deben desplegarse con un punto decimal. Esto normalmente se utiliza con **ios::fixed** para garantizar un cierto número de dígitos a la derecha del punto decimal. |
| **ios::uppercase** | Especifica que las letras mayúsculas (es decir, la **X** y de la **A** a la **F**) deben utilizarse en enteros hexadecimales, y que la letra mayúscula **E** debe utilizarse cuando se represente un valor de punto flotante en notación científica. |
| **ios::showpos** | Especifica que los números positivos y negativos deber estar precedidos por un signo + o −, respectivamente. |
| **ios::scientific** | Especifica la salida de un valor de punto flotante en notación científica. |
| **ios::fixed** | Especifica la salida de un valor de punto flotante en notación de punto fijo con un número específico de dígitos a la derecha del punto decimal. |

**Figura 21.20** Banderas de estado de formato. (Parte 2 de 2.)

Estas banderas pueden controlarse por medio de las funciones miembro **flags**, **setf** y **unsetf**, pero muchos de los programadores en C++ prefieren utilizar manipuladores de flujo (vea la sección 21.7.8). El programador puede utilizar la operación a nivel de bits or, |, para combinar varias opciones en un solo valor long (vea la figura 21.23). Llamar a la función miembro **flags** para un flujo, y especificar opciones separadas por medio de un or, ocasiona que se establezcan las opciones en ese flujo y que se devuelva un valor **long** que contenga las opciones anteriores. Con frecuencia el valor se guarda de modo que se puede llamar a **flags** con el valor guardado para restablecer las opciones previas de flujo.

La función **flags** debe especificar un valor que represente las configuraciones de todas las banderas. Por otra parte, la función **setf** con un argumento especifica una o más banderas separadas por un or, en donde cada una de ellas contiene las configuraciones existentes para formar un nuevo estado de formato.

El manipulador parametrizado de flujo **setiosflags** realiza las mismas funciones que la función miembro **setf**. El manipulador de flujo **resetiosflags** realiza las mismas funciones que la función miembro **unsetf**. Para utilizar cualquiera de estos manipuladores de flujo, asegúrese de incluir **#include<iomanip>**.

La bandera **skipws** indica que >> debe ignorar los espacios blancos de un flujo de entrada. El comportamiento predeterminado de >> es ignorar los espacios blancos. Para cambiar esto, utilice la llamada **unsetf (ios::skipws)**. El manipulador de flujo **ws** también puede utilizarse para especificar que los espacios blancos deben ignorarse.

## 21.7.2 Ceros a la derecha y puntos decimales (ios::showpoint)

La bandera **showpoint** se establece para forzar que un número de punto flotante se despliegue con su punto decimal y con ceros a la derecha. Un valor de punto flotante como **79.0** se imprimiría como **79** sin la bandera **showpoint**, y como **79.000000** (o con tantos ceros a la derecha como especifique la precisión actual) si se establece **showpoint**. El programa de la figura 21.21 muestra el uso de la función miembro **setf** para establecer la bandera **showpoint** y que se controlen los ceros a la derecha y la impresión del punto decimal para los valores de punto flotante.

```
// Figura 21.21: fig21_21.cpp
// Control de la impresión de ceros a la derecha y de
// puntos decimales con valores de punto flotante.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;

#include <cmath>

int main()
{
   cout << “Antes de establecer la bandera ios::showpoint\n”
        << “9.9900 se imprime como: “   << 9.9900
        << “\n9.9000 se imprime como: “ << 9.9000
        << “\n9.0000 se imprime como: “ << 9.0000
        << “\n\nDespues de establecer la bandera ios::showpoint\n”;
   cout.setf( ios::showpoint );
   cout << “9.9900 se imprime como: “   << 9.9900
        << “\n9.9000 se imprime como: “ << 9.9000
        << “\n9.0000 se imprime como: “ << 9.0000 << endl;
   return 0;
} // fin de la función main
```

```
Antes de establecer la bandera ios::showpoint
9.9900 se imprime como: 9.99
9.9000 se imprime como: 9.9
9.0000 se imprime como: 9

Despues de establecer la bandera ios::showpoint
9.9900 se imprime como: 9.99000
9.9000 se imprime como: 9.90000
9.0000 se imprime como: 9.00000
```

**Figura 21.21** Control de la impresión de ceros a la derecha y de puntos decimales con valores de punto flotante.

### 21.7.3 Justificación (`ios::left`, `ios::right`, `ios::internal`)

Las banderas **`left`** y **`right`** permiten que los campos se justifiquen a la izquierda con caracteres de relleno a la derecha, o que se justifiquen a la derecha con caracteres de relleno a la izquierda, respectivamente. El carácter que se utiliza como relleno lo especifica la función miembro **`fill`**, o el manipulador parametrizado de flujo **`setfill`** (vea la sección 21.7.4). La figura 21.22 muestra el uso de los manipuladores **`setw`**, **`setiosflags`** y **`resetiosflags`**, y las funciones miembro **`setf`** y **`unsetf`**, para controlar la justificación a la izquierda y a la derecha de datos enteros en un campo.

```
// Figura 21.22: fig21_22.cpp
// Justificación a la izquierda y a la derecha.
```

**Figura 21.22** Justificación a la izquierda y a la derecha. (Parte 1 de 2.)

```
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setw;
using std::setiosflags;
using std::resetiosflags;

int main()
{
   int x = 12345;

   cout << “La justificacion derecha es la predeterminada:\n”
        << setw(10) << x << “\n\nUSO DE LAS FUNCIONES MIEMBRO”
        << “\nUtilice setf para establecer ios::left:\n” << setw(10);

   cout.setf( ios::left, ios::adjustfield );
   cout << x << “\nUtilice unsetf para restablecer el valor
                  predeterminado:\n”;
   cout.unsetf( ios::left );
   cout << setw( 10 ) << x
        << “\n\nUSO DE LOS MANIPULADORES PARAMETRIZADOS DE FLUJO”
        << “\nUtilice setiosflags para establecer ios::left:\n”
        << setw( 10 ) << setiosflags( ios::left ) << x
        << “\nUtilice resetiosflags para restablecer el valor
           predeterminado:\n”
        << setw( 10 ) << resetiosflags( ios::left )
        << x << endl;
   return 0;
} // fin de la función main
```

```
La justificacion derecha es la predeterminada:
     12345

USO DE LAS FUNCIONES MIEMBRO
Utilice setf para establecer ios::left:
12345
Utilice unsetf para restablecer el valor predeterminado:
     12345
```

```
USO DE LOS MANIPULADORES PARAMETRIZADOS DE FLUJO
Utilice setiosflags para establecer ios::left:
12345
Utilice resetiosflags para restablecer el valor predeterminado:
     12345
```

**Figura 21.22** Justificación a la izquierda y a la derecha. (Parte 2 de 2.)

La bandera **`internal`** indica que el signo de un número (o base, cuando se establece la bandera **`ios::showbase`**; vea la sección 21.7.5) debe justificarse a la izquierda dentro de un campo, que la magni-

```
// Figura 21.23: fig21_23.cpp
// Impresión de un entero con espaciado interno y la impresión
// forzada del signo más.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setw;

int main()
{
   cout << setiosflags( ios::internal | ios::showpos )
        << setw( 10 ) << 123 << endl;
   return 0;
} // fin de la función main
```

```
+      123
```

**Figura 21.23** Impresión de un entero con espaciado interno y la impresión forzada del signo más.

tud del número debe justificarse a la derecha, y que los espacios intermedios deben rellenarse con el carácter de relleno. Las banderas **`left`**, **`right`** e **`internal`** se encuentran en el dato miembro estático **`ios::adjustfield`**. El argumento **`ios::adjustfield`** debe proporcionarse como el segundo argumento de **`setf`**, cuando se establecen las banderas de justificación **`left`**, **`right`** o **`internal`**. Esto permite a **`setf`** garantizar que se establezca sólo una de las tres banderas de justificación (éstas son mutuamente excluyentes). La figura 21.23 muestra el uso de los manipuladores **`setiosflags`** y **`setw`** para especificar el espaciado interno. Observe el uso de la bandera **`ios::showpos`** para forzar la impresión del signo más.

## 21.7.4 Relleno (`fill`, `setfill`)

La *función miembro* ***`fill`*** especifica el carácter de relleno a utilizarse en campos ajustados; si no se especifica valor alguno, se utilizan espacios como relleno. La función **`fill`** devuelve el carácter de relleno anterior. El *manipulador* ***`setfill`*** también establece el carácter de relleno. La figura 21.24 muestra el uso de la función miembro **`fill`** y del manipulador **`setfill`** para controlar la configuración y la reconfiguración del carácter de relleno.

```
// Figura 21.24: fig21_24.cpp
// Uso de la función miembro fill y del manipulador setfill
// para modificar el carácter de relleno, para
// campos más grandes que los valores a imprimirse.
#include <iostream>

using std::cout;
using std::endl;
```

**Figura 21.24** Uso de la función miembro **`fill`** y del manipulador **`setfill`** para modificar el carácter de relleno, para campos más grandes que los valores a imprimirse. (Parte 1 de 2.)

```

#include <iomanip>

using std::ios;
using std::setw;
using std::hex;
using std::dec;
using std::setfill;

int main()
{
   int x = 10000;

   cout << x << " impreso como un int justificado a izquierda y derecha\n"
        << "y como hex con justificacion interna.\n"
        << "Uso del caracter predeterminado de relleno (espacio):\n";
   cout.setf( ios::showbase );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << hex << x;

   cout << "\n\nUso de distintos caracteres de relleno:\n";
   cout.setf( ios::right, ios::adjustfield );
   cout.fill( '*' );
   cout << setw( 10 ) << dec << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << setfill( '%' ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
   return 0;
} // fin de la función main
```

```
10000 impreso como un int justificado a izquierda y derecha
y como hex con justificacion interna.
Uso del caracter predeterminado de relleno (espacio):
     10000
10000
0x    2710

Uso de distintos caracteres de relleno:
*****10000
10000%%%%%
0x^^^^2710
```

**Figura 21.24** Uso de la función miembro **fill** y del manipulador **setfill** para modificar el carácter de relleno, para campos más grandes que los valores a imprimirse. (Parte 2 de 2.)

### 21.7.5 Base de un flujo de enteros (ios::dec, ios::oct, ios::hex, ios::showbase)

El *miembro estático* ***ios::basefield*** (que se utiliza de manera similar a **ios::adjustfield** con **setf**) incluye los bits de banderas **ios::oct**, **ios::hex** e **ios::dec** para especificar que los enteros se

tratarán como valores octales, hexadecimales o decimales, respectivamente. Las inserciones de flujo predeterminadas son decimales, si no se establece uno de estos bits. El comportamiento predeterminado para las extracciones de flujo es que se procesen los datos en la forma en que éstos se proporcionan; los enteros que comienzan con 0 se tratan como valores octales, los enteros que comienzan con **`0x`** o **`0X`** se tratan como valores hexadecimales, y los demás enteros se tratan como valores decimales. Una vez que se especifica una base particular para un flujo, todos los enteros de ese flujo se procesan con esa base, hasta que se especifique una nueva base o hasta el final del programa.

Establezca la bandera **`showbase`** para forzar la impresión de la base de un valor entero. Los números decimales se despliegan de manera normal, los números octales se despliegan con un **`0`** a la izquierda, y los números hexadecimales se despliegan ya sea con un **`0x`** o con un **`0X`** a la izquierda (la bandera **`uppercase`** determina cuál opción es elegida; vea la sección 21.7.7). La figura 21.25 muestra el uso de la bandera **`showbase`** para forzar la impresión de un entero en formatos decimal, octal y hexadecimal.

```
// Figura 21.25: fig21_25.cpp
// Uso de la bandera ios::showbase.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::oct;
using std::hex;

int main()
{
   int x = 100;

   cout << setiosflags( ios::showbase )
        << "Impresion de enteros precedidos por su base:\n"
        << x << '\n'
        << oct << x << '\n'
        << hex << x << endl;
   return 0;
} // fin de la función main
```

```
Impresion de enteros precedidos por su base:
100
0144
0x64
```

**Figura 21.25** Uso de la bandera **`ios::showbase`**.

### 21.7.6 Números de punto flotante; notación científica (`ios::scientific`, `ios::fixed`)

Las *banderas* ***`ios::scientific`*** e ***`ios::fixed`*** se encuentran en el *dato miembro estático* ***`ios::floatfield`*** (estas banderas se utilizan de manera similar a **`ios::adjustfield`** e **`ios::basefield`** de **`setf`**). Estas banderas controlan el formato de salida de números de punto flotante. La bandera **`scientific`** se utiliza para forzar la impresión de un número de punto flotante en formato científico. La bandera **`fixed`** se utiliza para forzar la impresión de un número específico de dígitos (de acuerdo con lo especificado

por la función miembro **precision**) correspondientes a un número de punto flotante, a la derecha del punto decimal. Si no se establece una de estas banderas, el valor del número de punto flotante determina el formato de salida.

La llamada **cout.setf(0, ios::floatfield)** restablece el formato predeterminado para desplegar números de punto flotante. La figura 21.26 muestra la impresión de números de punto flotante en formatos fijo y científico, por medio de la función **setf** de dos argumentos con **ios::floatfield**. El formato del exponente de la notación científica puede variar entre compiladores.

```
// Figura 21.26: fig21_26.cpp
// Impresión de valores de punto flotante en formatos
// fijo, científico y el predeterminado por el sistema.
#include <iostream>

using std::cout;
using std::endl;
using std::ios;

int main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Desplegados con formato predeterminado:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::scientific, ios::floatfield );
   cout << "Desplegados con formato cientifico:\n"
        << x << '\t' << y << '\n';
   cout.unsetf( ios::scientific );
   cout << "Desplegados con formato predeterminado despues de unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::fixed, ios::floatfield );
   cout << "Desplegados con formato fijo:\n"
        << x << '\t' << y << endl;
   return 0;
} // fin de la función main
```

```
Desplegados con formato predeterminado:
0.00123457      1.946e+009
Desplegados con formato cientifico:
1.234567e-003   1.946000e+009
Desplegados con formato predeterminado despues de unsetf:
0.00123457      1.946e+009
Desplegados con formato fijo:
0.001235        1946000000.000000
```

**Figura 21.26** Impresión de valores de punto flotante en formatos fijo, científico y el predeterminado por el sistema.

## 21.7.7 Control de mayúsculas/minúsculas (ios::uppercase)

La bandera **ios::uppercase** fuerza la impresión de una **X** o una **E** mayúscula con los enteros hexadecimales o con valores de punto flotante en notación científica, respectivamente (figura 21.27). Cuando se establece, la bandera **ios::uppercase** ocasiona que todas las letras de un valor hexadecimal sean mayúsculas.

```
// Figura 21.27: fig21_27.cpp
// Uso de la bandera ios::uppercase
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setiosflags;
using std::ios;
using std::hex;

int main()
{
   cout << setiosflags( ios::uppercase )
        << “Impresion de letras mayusculas con notacion cientifica\n”
        << “con exponentes y valores hexadecimales:\n”
        << 4.345e10 << ‘\n’ << hex << 123456789 << endl;
   return 0;
} // fin de la función main
```

```
Impresion de letras mayusculas con notacion cientifica
con exponentes y valores hexadecimales:
4.345E+010
75BCD15
```

**Figura 21.27** Uso de la bandera **`ios::uppercase`**.

### 21.7.8 Cómo establecer y restablecer las banderas de formato (`flags`, `setiosflags`, `resetiosflags`)

La función miembro **`flags`** sin argumentos simplemente devuelve (como un valor **`long`**) la configuración actual de las banderas de formato. La función miembro **`flags`** con un argumento **`long`** establece las banderas de formato como lo especifique el argumento, y devuelve la configuración anterior. Cualquier bandera de formato no especificada en el argumento de **`flags`**, se restablecen. Observe que la configuración inicial de las banderas puede diferir de sistema a sistema. El programa de la figura 21.28 muestra el uso de la función miembro **`flags`** para establecer un nuevo estado de formato, y guarda el estado de formato anterior; después restablece las configuraciones de formato originales.

```
// Figura 21.28: fig21_28.cpp
// Demostración de la función miembro flags.
#include <iostream>

using std::cout;
using std::endl;
using std::ios;


int main()
{
   int i = 1000;
   double d = 0.0947628;
```

**Figura 21.28** Demostración de la función miembro **`flags`**. (Parte 1 de 2.)

```
    cout << "El valor de la variable flags es: "
         << cout.flags()
         << "\nImprime un int y un double con formato original:\n"
         << i << '\t' << d << "\n\n";
    long originalFormat =
            cout.flags( ios::oct | ios::scientific );
    cout << "El valor de la variable flags es: "
         << cout.flags()
         << "\nImprime int y double con un nuevo formato\n"
         << "especificado mediante el uso de la funcion miembro flags:\n"
         << i << '\t' << d << "\n\n";
    cout.flags( originalFormat );
    cout << "El valor de la variable flags es: "
         << cout.flags()
         << "\nImprime los valores de nuevo con el formato original:\n"
         << i << '\t' << d << endl;
    return 0;
} // fin de la función main
```

```
El valor de la variable flags es: 513
Imprime un int y un double con formato original:
1000    0.0947628

El valor de la variable flags es: 12000
Imprime int y double con un nuevo formato
especificado mediante el uso de la funcion miembro flags:
1750    9.476280e-002

El valor de la variable flags es: 513
Imprime los valores de nuevo con el formato original:
1000    0.0947628
```

**Figura 21.28** Demostración de la función miembro **`flags`**. (Parte 2 de 2.)

La función miembro **`setf`** establece las banderas de formato provistas en su argumento, y devuelve la configuración anterior como un valor **`long`**, como en

```
long ConfiguracionAnteriorBandera =
cout.setf( ios::showpoint | ios::showpos );
```

La función miembro **`setf`** con dos argumentos **`long`**, como en

```
cout.setf( ios::left, ios::adjustfield );
```

primero limpia los bits de **`ios:adjustfield`**, y después establece la bandera **`ios::left`**. Esta versión de **`setf`** se utiliza con los campos de bits asociados con **`ios::basefield`** (representado por **`ios::dec`**, **`ios::oct`** e **`ios::hex`**), **`ios::floatfield`** (representado por **`ios::scientific`** e **`ios::fixed`**) e **`ios::adjustfield`** (representado por **`ios::left`**, **`ios::right`** e **`ios::internal`**).

La función miembro **`unsetf`** restablece las banderas designadas, y devuelve el valor de las banderas, antes de que se restablezcan.

## 21.8 Estados de error de flujo

El estado de un flujo puede evaluarse a través de los bits de la clase **`ios`**; la clase base correspondiente a las clases **`istream`**, **`ostream`** e **`iostream`** que utilizamos para E/S.

El **eofbit** se establece para un flujo de entrada, después de que se encuentra el fin de archivo. Un programa puede utilizar la función miembro **eof** para determinar si el fin de archivo se encontró en el flujo, después de intentar extraer datos que se encuentran más allá del final del flujo. La llamada

```
cin.eof()
```

devuelve verdadero si se encontró el fin de archivo en **cin**; de lo contrario devuelve falso.

El **failbit** se establece en el flujo, cuando ocurre un error de formato. Por ejemplo, ocurre un error de formato cuando el programa introduce enteros y en el flujo encuentra un carácter que no es un dígito. Cuando ocurre dicho error, los caracteres no se pierden. La función miembro **fail** reporta si falló una operación del flujo; normalmente es posible recuperarse de tales errores.

El **badbit** se establece en un flujo, cuando ocurre un error que resulta en la pérdida de los datos. La función miembro **bad** reporta si falló una operación del flujo. Dichas fallas son serias, y normalmente no es posible recuperarse.

El **goodbit** se establece en un flujo, si no se establece alguno de los bits **eofbit**, **failbit** o **badbit**.

La función miembro **good** devuelve verdadero, si las funciones **bad**, **fail** y **eof** devuelven falso. Las operaciones de E/S sólo deben realizarse en flujos "buenos".

La función miembro **rdstate** devuelve el estado del error del flujo. Por ejemplo, una llamada a **cout.rdstate** devolvería el estado del flujo, el cual podría entonces evaluarse con una instrucción **switch** que examine **ios::eofbit**, **ios::badbit**, **ios::failbit** e **ios::goodbit**. Los medios preferidos para evaluar el estado de un flujo son las funciones miembro **eof**, **bad**, **fail** y **good**; para utilizar estas funciones, no es necesario que el programador esté familiarizado con un bit de estado en particular.

La función miembro **clear** normalmente se utiliza para restablecer a "bien" el estado de un flujo, de tal forma que la E/S pueda proceder en ese flujo. El argumento predeterminado para **clear** es **ios::goodbit**, de tal manera que la instrucción

```
cin.clear();
```

limpia **cin** y establece **goodbit** para el flujo. La instrucción

```
cin.clear( ios::failbit )
```

establece el **failbit**. El usuario podría desear hacer esto cuando realice una entrada en **cin** con un tipo definido por el usuario y se encuentre con un problema. El nombre **clear** puede parecer inapropiado en este contexto, sin embargo es correcto.

El programa de la figura 21.29 ilustra el uso de las funciones miembro **rdstate**, **eof**, **fail**, **bad**, **good** y **clear**. [*Nota:* Los valores reales de salida pueden diferir de compilador a compilador.]

```
// Figura 21.29: fig21_29.cpp
// Prueba de los estados de error.
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main()
{
   int x;
   cout << “Antes de una operacion de entrada incorrecta:”
        << “\ncin.rdstate(): “ << cin.rdstate()
        << “\n    cin.eof(): “ << cin.eof()
        << “\n   cin.fail(): “ << cin.fail()
```

**Figura 21.29** Prueba de los estados de error. (Parte 1 de 2.)

```
          << “\n    cin.bad(): “ << cin.bad()
          << “\n   cin.good(): “ << cin.good()
          << “\n\nEspera un entero, pero se introduce un caracter: “;
     cin >> x;

     cout << “\nDespues de una operacion incorrecta:”
          << “\ncin.rdstate(): “ << cin.rdstate()
          << “\n    cin.eof(): “ << cin.eof()
          << “\n   cin.fail(): “ << cin.fail()
          << “\n    cin.bad(): “ << cin.bad()
          << “\n   cin.good(): “ << cin.good() << “\n\n”;

     cin.clear();

     cout << “Despues de cin.clear()”
          << “\ncin.fail(): “ << cin.fail()
          << “\ncin.good(): “ << cin.good() << endl;
     return 0;
} // fin de la función main
```

```
Antes de una operacion de entrada incorrecta:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Espera un entero, pero se introduce un caracter: A

Despues de una operacion incorrecta:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
    cin.bad(): 0
   cin.good(): 0

Despues de cin.clear()
cin.fail(): 0
cin.good(): 1
```

**Figura 21.29** Prueba de los estados de error. (Parte 2 de 2.)

La función miembro **`operator!`** devuelve verdadero si **`badbit`** o **`failbit`** se establece, o si se establecen ambas. La función miembro **`operator void`** * devuelve falso (0) si se establece **`badbit`** o **`failbit`**, o ambas. Estas funciones son útiles en el procesamiento de archivos, cuando se evalúa una condición verdadera/falsa bajo el control de una estructura de selección o de repetición.

## 21.9 Unión de un flujo de salida con un flujo de entrada

Las aplicaciones interactivas generalmente involucran un **`istream`** para la entrada de datos y un **`ostream`** para la salida. Cuando aparece un mensaje de indicaciones en la pantalla, el usuario responde introduciendo los datos apropiados. Obviamente, las indicaciones deben aparecen antes de que la operación de entrada proceda. Con una salida con búfer, ésta sólo aparece cuando el búfer se llena, cuando las salidas son vaciadas explícitamente por el programa, o automáticamente al final del programa. C++ proporciona la función miembro **`tie`**

para sincronizar (es decir, para "unir") la operación de un **istream** y un **ostream**, para garantizar que las salidas aparezcan antes de sus entradas subsiguientes. La llamada

```
cin.tie( &cout );
```

une **cout** (un **ostream**) con **cin** (un **istream**). De hecho, esta llamada en particular es redundante, ya que C++ realiza automáticamente esta operación para crear un ambiente estándar de usuario de entrada/salida. Sin embargo, el usuario uniría explícitamente otro par de **istream/ostream**. Para desunir un flujo de entrada, **inputStream**, de un flujo de salida, utilice la llamada

```
inputStream.tie( 0 );
```

## RESUMEN

- Las operaciones de E/S se realizan de una manera sensible al tipo de los datos.
- Las E/S en C++ ocurren en flujos de bytes. Un flujo es simplemente una secuencia de bytes.
- Los mecanismos de E/S del sistema mueven los bytes desde los dispositivos hacia la memoria y viceversa, de una manera eficiente y confiable.
- C++ proporciona capacidades de E/S de "bajo nivel" y de "alto nivel". Las capacidades de E/S de bajo nivel especifican que cierto número de bytes deben transferirse desde un dispositivo hacia la memoria, o desde la memoria hacia un dispositivo. Las E/S de alto nivel se realizan con bytes agrupados en unidades significativas como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por el usuario.
- C++ proporciona operaciones de E/S con formato y sin formato. Las transferencias de E/S sin formato son rápidas, pero se procesan datos puros que a la gente se le dificulta utilizar. La E/S con formato procesa datos en unidades significativas, pero requieren tiempo de procesamiento adicional que puede afectar negativamente las transferencias de grandes volúmenes de datos.
- La mayoría de los programas en C++ incluyen el archivo de encabezado **<iostream>** que declara todas las operaciones de E/S de flujos.
- El encabezado **<iomanip>** declara la entrada/salida con formato con manipuladores parametrizados de flujo.
- El encabezado **<fstream>** declara operaciones de procesamiento de archivos.
- La clase **istream** soporta operaciones de entrada de flujo.
- La clase **ostream** soporta operaciones de salida de flujo.
- La clase **iostream** soporta operaciones de entrada y de salida de flujo.
- Las clases **istream** y **ostream** se derivan a través de la herencia simple desde la clase base **ios**.
- La clase **iostream** se deriva a través de la herencia múltiple desde las clases **istream** y **ostream**.
- El operador de desplazamiento a la izquierda (<<) se sobrecarga para designar la salida de flujo, y se le conoce como operador de inserción de flujo.
- El operador de desplazamiento a la derecha (>>) se sobrecarga para designar la entrada de flujo, y se le conoce como operador de extracción de flujo.
- El objeto **cin** de **istream** está unido con el dispositivo de entrada estándar, que por lo general es el teclado.
- El objeto **cout** de la clase **ostream** está unido con el dispositivo de salida estándar, que por lo general es la pantalla.
- El objeto **cerr** de la clase **ostream** está unido con el dispositivo de error estándar. Las salidas de **cerr** son sin búfer; cada inserción de **cerr** aparece de inmediato.
- El manipulador de flujo **endl** despliega un carácter de nueva línea y vacía el búfer de salida.
- El compilador de C++ determina automáticamente los datos de entrada y salida.
- Las direcciones se despliegan de manera predeterminada en formato hexadecimal.
- Para imprimir la dirección de una variable apuntador, realice la conversión de tipo del apuntador a **void** *.
- La función miembro **put** despliega un carácter. Las llamadas a **put** pueden ser en cascada.
- La entrada de flujo se realiza con el operador de extracción de flujo >>. Este operador automáticamente ignora los caracteres blancos del flujo de entrada.
- El operador >> devuelve falso, después de que se encuentra el fin de archivo en un flujo.
- La extracción de flujo ocasiona que se establezca el **failbit** para entradas inadecuadas, y **badbit** si la operación falla.

- Es posible introducir una serie de valores por medio de la operación de extracción de flujo en un encabezado de ciclo **`while`**. La extracción devuelve 0, cuando se encuentra el fin de archivo.
- La función **`get`** sin argumentos introduce un carácter y lo devuelve; si el fin de archivo se encuentra en el flujo, se devuelve **`EOF`**.
- La función miembro **`get`** con un argumento de tipo referencia a un **`char`** introduce un carácter. Cuando se encuentra el fin de archivo, se devuelve **`EOF`**; de lo contrario, se devuelve el objeto **`istream`** para el que se invocó a la función miembro **`get`**.
- La función **`get`** con tres argumentos (un arreglo de caracteres, un límite de tamaño y un delimitador con el valor predeterminado de nueva línea) lee los caracteres desde el flujo de entrada hasta un máximo de límite de un carácter y finaliza, o termina cuando lee el delimitador. La cadena de entrada termina con un carácter nulo. El delimitador no se coloca en el arreglo de caracteres, pero permanece en el flujo de entrada.
- La función miembro **`getline`** opera como la función miembro **`get`** de tres argumentos. La función **`getline`** elimina el delimitador del flujo de entrada, pero no lo almacena en la cadena.
- La función miembro **`ignore`** pasa por alto el número especificado de caracteres (el predeterminado es 1) en el flujo de entrada; ésta termina si encuentra el delimitador especificado (el predeterminado es **`EOF`**).
- La función miembro **`putback`** coloca el carácter previamente obtenido en un flujo por un **`get`**, de regreso en ese flujo.
- La función miembro **`peek`** devuelve el siguiente carácter de un flujo de entrada, pero no extrae (elimina) el carácter del flujo.
- C++ ofrece E/S con seguridad de tipos. Si se procesan datos inesperados con los operadores << y >>, se establecen varias banderas de error, las cuales utiliza el usuario para determinar si una operación de E/S se realizó con éxito, o si falló.
- La E/S sin formato se realiza con las funciones miembro **`read`** y **`write`**. Éstas introducen o despliegan cierto número de bytes hacia o desde la memoria, comenzando en una dirección de memoria designada. Éstos se despliegan como bytes puros sin formato alguno.
- La función miembro **`gcount`** devuelve el número de caracteres introducidos en ese flujo por la operación **`read`** anterior.
- La función miembro **`read`** introduce un número especificado de caracteres en un arreglo de caracteres. Si se leen menos caracteres que el número especificado, se establece **`failbit`**.
- Para modificar la base en la que se despliegan los enteros, utilice el manipulador **`hex`** para establecer la base en hexadecimal, **`oct`** para establecer la base en octal (base 8). Utilice el manipulador **`dec`** para restablecer la base en decimal. La base permanece igual hasta que explícitamente se modifique.
- El manipulador parametrizado de flujo **`setbase`** también establece la base para la salida de enteros. Para establecer la base, **`setbase`** toma un argumento entero de **`10`**, **`8`** o **`16`**.
- La precisión de un número de punto flotante puede controlarse por medio del manipulador de flujo **`setprecision`** o por medio de la función miembro **`precision`**. Ambos establecen la precisión de todas las operaciones de salida subsiguientes, hasta la siguiente llamada para establecer otra precisión. La función miembro **`precision`** sin argumentos devuelve el valor de la precisión actual.
- Los manipuladores parametrizados requieren la inclusión del archivo de encabezado <**`iomanip`**>.
- La función miembro **`width`** establece el ancho del campo y devuelve el ancho anterior. Los valores más pequeños que el campo se rellenan con caracteres de relleno. La configuración del ancho de campo aplica sólo para la siguiente inserción o extracción; después, el ancho de campo se establece implícitamente en **`0`** (los valores subsiguientes se desplegarán tan grandes como sea necesario). Los valores mayores que un campo se imprimen en su totalidad. La función **`width`** sin argumentos devuelve la configuración actual del ancho de campo. El manipulador **`setw`** también establece el ancho del campo.
- Para entrada, el manipulador de flujo **`setw`** establece un tamaño de cadena máximo; si se introduce una cadena más grande, la línea más grande se parte en piezas no mayores que el tamaño designado.
- Los usuarios pueden crear sus propios manipuladores de flujo.
- Las funciones miembro **`setf`**, **`unsetf`** y **`flags`** controlan las configuraciones de las banderas.
- La bandera **`skipws`** indica que >> debe ignorar los caracteres blancos en un flujo de entrada. El manipulador de flujo **`ws`** también ignora los caracteres blancos a la izquierda de un flujo de entrada.
- Las banderas de formato se definen como una enumeración en la clase **`ios`**.
- Las banderas de formato se controlan con las funciones miembro **`flags`** y **`setf`**, pero muchos programadores en C++ prefieren utilizar manipuladores de flujo. La operación a nivel de bits **`or`**, **`|`**, puede utilizarse para combinar varias op-

ciones en un solo valor **long**. Llamar a la función miembro **flags** para un flujo, y especificar estas opciones separadas por **or**, establece las opciones en ese flujo y devuelve un valor **long** que contiene las opciones anteriores. Este valor con frecuencia se guarda para que **flags** pueda ser llamada con este valor para restablecer las opciones anteriores del flujo.

- La función **flags** debe especificar un valor que represente todas las configuraciones de todas las banderas. Por otra parte, la función **setf** con un argumento automáticamente separa con **or** las banderas especificadas con las configuraciones de bandera existentes, para formar un nuevo estado de formato.
- La bandera **showpoint** se establece para forzar a que un número de punto flotante se despliegue con un punto decimal y un número significativo de dígitos, especificados por la precisión.
- Las banderas **left** y **right** ocasionan que los campos se justifiquen a la izquierda con caracteres de relleno a la derecha, o que se justifiquen a la derecha con caracteres de relleno a la izquierda.
- La bandera **internal** indica que el signo de un número (o una base, cuando se establece la bandera **ios::showbase**) debe justificarse a la izquierda dentro de un campo, que la magnitud debe justificarse a la derecha, y que los espacios intermedios deben rellenarse con el carácter de relleno.
- **ios::adjustfield** contiene las banderas **left**, **right** e **internal**.
- La función miembro **fill** especifica el carácter de relleno a usarse con los campos ajustados **left**, **right** e **internal** (el predeterminado es el espacio); se devuelve el carácter de relleno anterior. El manipulador de flujo **setfill** también establece el carácter de relleno.
- El miembro estático **ios::basefield** tiene los bits **oct**, **hex** y **dec** para especificar que los enteros van a tratarse como valores octales, hexadecimales o decimales, respectivamente. La salida de enteros predeterminada es en decimal, si ninguno de estos bits se establece; las extracciones de flujo procesan los datos en la forma en que éstos se proporcionan.
- Establezca la bandera ***showbase*** para forzar a que se despliegue la base de un valor entero.
- El dato miembro estático **ios::floatfield** contiene las banderas **scientific** y **fixed**. Establezca la bandera **scientific** para desplegar un número de punto flotante en formato científico. Establezca la bandera **fixed** para desplegar un número de punto flotante con la precisión especificada por la función miembro **precision**.
- La llamada a **cout.setf**(0, **ios::floatfield**) restablece el formato predeterminado para desplegar números de punto flotante.
- Establezca la bandera **uppercase** para forzar a que se despliegue una **X** o una **E** mayúscula con enteros hexadecimales o valores de punto flotante en notación científica, respectivamente. Cuando se establece, la bandera **ios::uppercase** ocasiona que todas las letras de un valor hexadecimal sean mayúsculas.
- La función miembro **flags** sin argumentos devuelve el valor **long** de las configuraciones actuales de las banderas de formato. La función miembro **flags** con un argumento **long** establece las banderas de formato especificadas por el argumento, y devuelve las configuraciones de bandera anteriores.
- La función miembro **setf** establece las banderas de formato en su argumento, y devuelve las configuraciones anteriores como un valor **long**.
- La función miembro **setf**( **long setBits**, **long resetBits**) limpia los bits de **resetBits**, y después establece el **bit** en **setBits**.
- La función miembro **unsetf** restablece las banderas designadas y devuelve el valor anterior de las banderas.
- El manipulador parametrizado de flujo **setiosflags** realiza las mismas funciones que la función miembro **flags**.
- El manipulador parametrizado de flujo **resetiosflags** realiza las mismas funciones que la función miembro **unsetf**.
- El estado de un flujo puede evaluarse por medio de los bits de la clase **ios**.
- El **eofbit** se establece para un flujo de entrada, después de que se encuentra el fin de archivo durante una operación de entrada. La función miembro **eof** reporta si se estableció el **eofbit**.
- El **failbit** se establece en un flujo, cuando ocurre un error de formato en dicho flujo. Ningún carácter se pierde. La función miembro **fail** reporta si una operación de flujo falló; normalmente es posible recuperarse de tales errores.
- El **badbit** se establece en un flujo, cuando ocurre un error que resulta en la pérdida de los datos. La función miembro **bad** reporta si una operación de flujo falló. Normalmente no es posible recuperarse de estos serios errores.
- La función miembro **good** devuelve verdadero, si las funciones **bad**, **fail** y **eof** devuelven falso. Las operaciones de E/S sólo deben realizarse en flujos “buenos”.
- La función miembro **rdstate** devuelve el estado del error del flujo.
- La función miembro **clear** normalmente se utiliza para restablecer el estado de un flujo en “bueno”, para que la E/S pueda proceder en ese flujo.

- C++ proporciona la función miembro **tie** para sincronizar operaciones **istream** y **ostream**, para garantizar que las salidas aparezcan antes de las entradas subsiguientes.

## TERMINOLOGÍA

**0** a la izquierda (octal)
**0x** o **0X** a la izquierda (hexadecimal)
ancho
ancho de campo
archivo de encabezado estándar <**iomanip**>
**badbit**
banderas de formato
carácter de relleno
carácter predeterminado de relleno (espacio)
caracteres blancos
**cerr**
**cin**
clase **fstream**
clase **ifstream**
clase **ios**
clase **iostream**
clase **istream**
clase **ofstream**
clase **ostream**
**clog**
**cout**
E/S con formato
E/S con seguridad de tipos
E/S sin formato
**endl**
entrada de flujo
**eofbit**
estados de formato
**failbit**
fin de archivo
flujos definidos por el usuario
flujos predefinidos
función miembro **bad**
función miembro **clear**
función miembro **eof**
función miembro **fail**
función miembro **fill**
función miembro **flags**
función miembro **flush**
función miembro **gcount**
función miembro **get**
función miembro **getline**
función miembro **good**
función miembro **ignore**
función miembro **operator void***
función miembro **operator!**
función miembro **peek**
función miembro **precision**
función miembro **put**
función miembro **putback**
función miembro **rdstate**
función miembro **read**
función miembro **setf**
función miembro **tie**
función miembro **unsetf**
función miembro **write**
función miembro **ws**
**ios::adjustfield**
**ios::basefield**
**ios::fixed**
**ios::floatfield**
**ios::internal**
**ios::scientific**
**ios::showbase**
**ios::showpoint**
**ios::showpos**
justificación a la derecha
justificación a la izquierda
manipulador de flujo **dec**
manipulador de flujo **flush**
manipulador de flujo **hex**
manipulador de flujo **oct**
manipulador de flujo **resetiosflags**
manipulador de flujo **setbase**
manipulador de flujo **setfill**
manipulador de flujo **setiosflags**
manipulador de flujo **setprecision**
manipulador de flujo **setw**
manipulador parametrizado de flujo
manipulador **stream**
mayúscula
operador de extracción de flujo (>>)
operador de inserción de flujo (<<)
precisión predeterminada
relleno
salida de flujo
**skipws**

## ERRORES COMUNES DE PROGRAMACIÓN

**21.1** Intentar realizar una lectura desde un **ostream** (o desde cualquier otro flujo de sólo salida), es un error.

**21.2** Intentar escribir en un **istream** (o en cualquier otro flujo de sólo entrada), es un error.

**21.3** No proporcionar paréntesis para forzar la precedencia adecuada, cuando se utiliza la relativa alta precedencia del operador de inserción de flujo << o del operador de extracción de flujo >>, es un error.

**21.4** Un ancho establecido aplica sólo para la siguiente inserción o extracción; después de eso, el ancho se establece implícitamente en **0** (es decir, los valores desplegados simplemente serán tan amplios como sea necesario). La función **width** sin argumentos devuelve el valor establecido actual. Asumir que el ancho establecido se aplica a todas las salidas subsiguientes, es un error lógico.

**21.5** Cuando no proporciona un ancho de campo suficiente para manejar las salidas, éstas se imprimen tan amplias como sea necesario, lo que probablemente ocasione dificultades para leerlas.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**21.1** En programas de C++ utilice exclusivamente la forma de E/S de C++, aunque el estilo de C para E/S esté disponible para los programadores en C++.

**21.2** Cuando despliegue expresiones, colóquelas entre paréntesis para evitar problemas con la precedencia de los operadores de la expresión y el operador <<.

## TIP DE RENDIMIENTO

**21.1** Utilice E/S sin formato, para un mejor rendimiento en el procesamiento de archivos de gran volumen.

## TIP DE PORTABILIDAD

**21.1** Cuando indique al usuario cómo terminar la introducción de datos desde el teclado, solicítele que "introduzca el fin de archivo para finalizar la entrada de datos", en lugar de solicitarle un *<ctrl>d* (UNIX y Macintosh) o *<ctrl>z* (PC y VAX).

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**21.1** El estilo de E/S de C++ ofrece seguridad de tipos.

**21.2** C++ ofrece un tratamiento común de E/S de tipos predefinidos y de tipos definidos por el usuario. Este tipo de tratamiento común facilita el desarrollo de software en general y la reutilización de software en particular.

## EJERCICIOS DE AUTOEVALUACIÓN

**21.1** Complete los espacios:

a) Los operadores de flujo sobrecargados con frecuencia se definen como funciones __________ de una clase.
b) Los bits para justificación de formato que pueden establecerse incluyen __________, __________ y __________.
c) En C++, la E/S ocurre como __________ de bytes.
d) Los manipuladores parametrizados de flujo __________ y __________ pueden utilizarse para establecer y restablecer banderas de estado de formato.
e) La mayoría de los programas en C++ deben incluir el archivo de encabezado __________ que contiene las declaraciones requeridas para todas las operaciones de E/S de flujo.
f) Las funciones miembro __________ y __________ establecen y restablecen banderas de estado de formato.
g) El archivo de encabezado __________ contiene las declaraciones necesarias para realizar un formato "en memoria".
h) Cuando se utilizan manipuladores parametrizados, debe incluirse el archivo de encabezado __________.
i) El encabezado __________ contiene las declaraciones requeridas para el procesamiento de archivos controlado por el usuario.
j) El manipulador de flujo __________ inserta un carácter de nueva línea en el flujo de salida y vacía el flujo de salida.
k) El archivo de encabezado __________ se utiliza en programas que mezclan en estilo de E/S de C y de C++.
l) La función miembro de **`ostream`** __________ se utiliza para realizar salidas sin formato.
m) Las operaciones de entrada son soportadas por la clase __________.
n) Las salidas del flujo de error estándar son dirigidas hacia el objeto de flujo __________ o __________.
o) Las operaciones de salida son soportadas por la clase __________.
p) El símbolo para el operador de inserción de flujo es __________.
q) Los cuatro objetos que corresponden a los dispositivos estándar del sistema incluyen __________, __________, __________ y __________.
r) El símbolo para el operador de extracción de flujo es __________.
s) Los manipuladores de flujo __________, __________ y __________ especifican que los enteros deben desplegarse en formato octal, hexadecimal y decimal, respectivamente.
t) La precisión predeterminada para desplegar valores de punto flotante es __________.
u) Cuando se establece, la bandera __________ ocasiona que los números positivos se desplieguen con un signo más.

**21.2** Establezca si los siguientes son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.

a) La función miembro **`flags`**() con un argumento **`long`** establece a la variable de estado **`flags`** en su argumento, y devuelve su valor anterior.

b) El operador de inserción de flujo << y el operador de extracción de flujo >> se sobrecargan para manejar todos los tipos estándar, incluso cadenas y direcciones de memoria (sólo inserción de flujo), y todos los tipos de datos definidos por el usuario.
c) La función miembro **flags()** sin argumentos restablece todos los bits de bandera en la variable de estado banderas.
d) El operador de extracción de flujo >> puede sobrecargarse con una función de operador que toma como argumentos una referencia **istream** y una referencia hacia un tipo definido por el usuario, y devuelve una referencia **istream**.
e) El manipulador de flujo **ws** ignora espacios blancos a la izquierda de un flujo de entrada.
f) El operador de inserción de flujo << puede sobrecargarse con una función operador que toma como argumentos una referencia **istream** y una referencia hacia un tipo definido por el usuario, y devuelve una referencia **istream**.
g) La entrada con el operador de extracción de flujo >> siempre ignora los espacios blancos a la izquierda del flujo de entrada.
h) Las características de entrada y de salida se proporcionan como parte de C++.
i) La función miembro **rdstate()** devuelve el estado actual del flujo.
j) El flujo **cout** normalmente está conectado a la pantalla.
k) La función miembro **good** devuelve verdadero, si las funciones miembro **bad()**, **fail()** y **eof()** devuelven falso.
l) El flujo **cin** normalmente está conectado a la pantalla.
m) Si ocurre un error no recuperable durante una operación de flujo, la función miembro **bad** devolverá verdadero.
n) La salida de **cerr** es sin búfer, y la salida con **clog** es con búfer.
o) Cuando se establece la bandera **ios::showpoint**, los valores de punto flotante son forzados a imprimirse con los seis dígitos de precisión predeterminada; dado que el valor de la precisión so se ha modificado, los valores de punto flotante se imprimen con la precisión especificada.
p) La función miembro de **ostream put** despliega el número especificado de caracteres.
q) Los manipuladores de flujo **dec**, **oct** y **hex** sólo afectan la siguiente operación de salida de enteros.
r) Cuando se despliegan, las direcciones de memoria aparecen de manera predeterminada como enteros **long**.

**21.3** Para cada uno de los siguientes, escriba una sola instrucción que realice la tarea indicada.
a) Despliegue la cadena "**Escriba su nombre:** ".
b) Establezca una cadena que ocasione que el exponente de la notación científica y que las letras de valores hexadecimales se impriman en letras mayúsculas.
c) Despliegue la dirección de la variable cadena de tipo **char***.
d) Establezca una bandera para que los valores de punto flotante se impriman en notación científica.
e) Despliegue la dirección de la variable **ptrEntero** de tipo **int***.
f) Establezca una bandera para que cuando se desplieguen valores enteros, se despliegue la base de los enteros octales y hexadecimales.
g) Despliegue el valor al que apunta **ptrFlotante** de tipo **float***.
h) Utilice una función miembro de flujo para establecer en '*' al carácter de relleno, para que se imprima en anchos de campo mayores que los valores a desplegar. Escriba una instrucción separada que haga esto con un manipulador de flujo.
i) Despliegue los caracteres **'O'** y **'K'** en una instrucción con la función **put** de **ostream**.
j) Obtenga el valor del siguiente carácter del flujo de entrada, sin extraerlo del flujo.
k) Introduzca un solo carácter dentro de la variable **c** de tipo **char**, por medio de la función miembro ***get*** de **istream** en dos formas diferentes.
l) Introduzca y descarte los siguientes seis caracteres de un flujo de entrada.
m) Utilice la función miembro ***read*** de ***istream*** para introducir 50 caracteres en un arreglo ***linea*** de tipo ***char***.
n) Lea 10 caracteres del arreglo de caracteres nombre. Detenga la lectura si se encuentra el delimitador **'.'**. No elimine el delimitador del flujo de entrada. Escriba otra instrucción que realice esta tarea y que remueva el delimitador de la entrada.
o) Utilice la función miembro **gcount** de **istream**, para determinar el número de caracteres introducidos en el arreglo de caracteres **linea** por medio de la última llamada a la función miembro **read** de **istream**, y despliegue ese número de caracteres a través de la función miembro **write** de **ostream**.
p) Escriba instrucciones separadas para vaciar el flujo de salida por medio de una función miembro y de un manipulador de flujo.
q) Despliegue los siguientes valores: **124**, **18.376**, **'Z'**, **1000000**, y **"Cadena"**.

r) Imprima la configuración actual de la precisión por medio de una función miembro.
s) Introduzca un valor entero dentro de la variable **int** meses, y un valor de punto flotante en la variable **float tasaPorcentual**.
t) Por medio de un manipulador, imprima **1.92**, **1.925** y **1.9258** con tres dígitos de precisión.
u) Por medio de manipuladores de flujo, imprima el entero **100** en formato octal, hexadecimal y decimal.
v) Imprima el entero **100** en formato decimal, octal y hexadecimal, utilizando un solo manipulador de flujo para cambiar la base.
w) Imprima **1234** justificado a la derecha, en un campo de **10** dígitos.
x) Lea los caracteres del arreglo **linea**, hasta que se encuentre el carácter **'z'**, en un límite de **20** caracteres (que incluya el carácter de terminación nulo). No extraiga el carácter delimitador del flujo.
y) Utilice las variables enteras **x** y **y** para especificar el ancho de un campo y la precisión utilizada para desplegar el valor **double 87.4573**, e imprima el valor.

**21.4** Identifique el error en cada una de las siguientes instrucciones, y explique cómo corregirlo.

a) **cout << "El valor de x <= y es: " << x <= y;**
b) La siguiente instrucción debe desplegar el valor de **'c'**.
**cout << 'c';**
c) **cout << ""Una cadena entre comillas"";**

**21.5** Para cada una de las siguientes, muestre la salida.

a)
```
cout << "12345" << endl;
cout.width( 5 );
cout.fill( '*' );
cout << 123 << endl << 123;
```
b) **cout << setw( 10 ) << setfill( '$' ) << 10000;**
c) **cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;**
d) **cout << setiosflags( ios::showbase ) << oct << 99 << endl << hex << 99;**
e)
```
cout << 100000 << endl
           << setiosflags( ios::showpos ) << 100000;
```
f)
```
cout << setw( 10 ) << setprecision( 2 ) <<
            setiosflags( ios::scientific ) << 444.93738;
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**21.1** a) **friend**. b) **ios::left**, **ios::right** e **ios::internal**. c) Flujos. d) **setiosflags**, **resetiosflags**. e) **iostream**. f) **setf**, **unsetf**. g) **strstream**. h) **iomanip**. i) **fstream**. j) **endl**. k) **stdiostream**. l) **write**. m) **istream**. n) **cerr** o **clog**. o) **ostream**. p) **<<**. q) **cin**, **cout**, **cerr** y **clog**. r) **>>**. s) **oct**, **hex** y **dec**. t) Seis dígitos de precisión. u) **ios::showpos**.

**21.2** a) Verdadero.
b) Falso. Los operadores de inserción y de extracción de flujo no se sobrecargan para todos los tipos definidos por el usuario. El programador de una clase debe proporcionar específicamente las funciones de operador sobrecargadas, para sobrecargar los operadores de flujo para utilizarlos con cada tipo definido por el usuario.
c) Falso. La función miembro de flujo **flags()** sin argumentos simplemente devuelve el valor actual de la variable de estado **flags**.
d) Verdadero.
e) Verdadero.
f) Falso. Para sobrecargar el operador de inserción de flujo <<, la función de operador sobrecargada debe tomar como argumentos una referencia **ostream** y una referencia a un tipo definido por el usuario, y devuelve una referencia **ostream**.
g) Verdadero. A menos que **ios::skipws** esté desactivado.
h) Falso. Las características de E/S de C++ se proporcionan como parte de la Biblioteca Estándar de C++. El lenguaje C++ no contiene capacidades para entrada, salida, o procesamiento de archivos.
i) Verdadero.
j) Verdadero.
k) Verdadero.
l) Falso. El flujo **cin** está conectado a la entrada estándar de la computadora, la cual normalmente es el teclado.
m) Verdadero.
n) Verdadero.

o) Verdadero.
p) Falso. La función miembro **put** de **ostream** despliega su argumento de un solo carácter.
q) Falso. Los manipuladores de flujo **dec**, **oct** y **hex** establecen el estado de formato de salida para enteros con la base especificada, a menos que la base se modifique nuevamente o que el programa termine.
r) Falso. Las direcciones de memoria se despliegan de manera predeterminada en formato hexadecimal. Para desplegar direcciones como enteros **long**, éstas deben convertirse al tipo de un valor **long**.

**21.3** a) **cout << "Escriba su nombre: ";**
b) **cout.setf(ios::uppercase);**
c) **cout << (void *) cadena;**
d) **cout.setf(ios::scientific, ios::floatfield);**
e) **cout << ptrEntero;**
f) **cout << setiosflags(ios::showbase);**
g) **cout << *ptrFlotante;**
h) **cout.fill( '*' );**
**cout << setfill( '*' );**
i) **cout.put( 'O' ).put( 'K' );**
j) **cin.peek();**
k) **c = cin.get();**
**cin.get( c );**
l) **cin.ignore( 6 );**
m) **cin.read( linea, 50 );**
n) **cin.get( nombre, 10, '.' );**
**cin.getline( nombre, 10, '.' );**
o) **cout.write( linea, cin.gcount() );**
p) **cout.flush();**
**cout << flush;**
q) **cout << 124 << 18.376 << 'Z' << 1000000 << "Cadena";**
r) **cout << cout.precision();**
s) **cin >> meses >> tasaPorcentual;**
t) **cout << setprecision( 3 ) << 1.92 << '\t'**
**<< 1.925 << '\t' << 1.9258;**
u) **cout << oct << 100 << hex << 100 << dec << 100;**
v) **cout << 100 << setbase( 8 ) << 100 << setbase( 16 ) << 100 ;**
w) **cout << setw( 10 ) << 1234 ;**
x) **cin.get( linea, 20, 'z' );**
y) **cout << setw( x ) << setprecision( y ) << 87.4573 ;**

**21.4** a) Error: la precedencia del operador << es más alta que la precedencia de <=, lo cual ocasiona que la instrucción se evalúe inadecuadamente, y también ocasiona un error de compilación.
Corrección: para corregir la instrucción, agregue paréntesis alrededor de la expresión **x** <= **y**. Este problema ocurrirá con cualquier expresión que utilice operadores de precedencia más baja que el operador <<, si la expresión no se coloca entre paréntesis.
b) Error: en C++, los caracteres no se tratan como enteros pequeños, como sucede en C.
Corrección: para imprimir el valor numérico de un carácter del conjunto de caracteres de la computadora, éste debe convertirse al tipo de un valor entero de la siguiente forma:

```
cout << int( 'c' );
```

c) Error: los caracteres comillas no pueden imprimirse en una cadena, a menos que se utilice una secuencia de escape.
Corrección: imprima la cadena en una de las siguientes formas:

```
cout << '"' << "Una cadena entre comillas" << '"';
cout << "\"Una cadena entre comillas\"";
```

**21.5** a)
```
12345
  **123
  123
```
b)
```
$$$$$100000
```

c) **1024.988**
d) **0143**
   **0x63**
e) **100000**
   **+100000**
f) **4.45e+02**

## EJERCICIOS

**21.6** Escriba una instrucción para cada una de las siguientes tareas:
a) Imprima el entero **40000** justificado a la izquierda en un campo de **15** dígitos.
b) Lea una cadena dentro del arreglo de caracteres **estado**.
c) Imprima **200** con y sin signo.
d) Imprima el valor decimal **100** en formato hexadecimal precedido por **0x**.
e) Lea los caracteres del arreglo s, hasta que encuentre **'p'** en un límite de 10 caracteres (que incluye al carácter de terminación nulo). Extraiga el delimitador del flujo de entrada y descártelo.
f) Imprima **1.234** en un campo de 9 dígitos con ceros a la izquierda.
g) Lea una cadena de la forma **"caracteres"** desde la entrada estándar. Almacene la cadena en el arreglo de caracteres **s**. Elimine las comillas del flujo de entrada. Lea un máximo de **50** caracteres (que incluyan el carácter de terminación nulo).

**21.7** Escriba un programa que evalúe la entrada de valores enteros en formato decimal, octal y hexadecimal. Despliegue cada carácter leído por el programa en los tres formatos. Evalúe el programa con los siguientes datos de entrada: **10**, **010**, **0x10**.

**21.8** Escriba un programa que imprima valores de apuntador, utilizando conversiones de tipo para todos los tipos de datos enteros. ¿Cuál de ellos imprime valores extraños? ¿Cuál ocasiona errores?

**21.9** Escriba un programa que evalúe los resultados de imprimir el valor entero **12345** y el valor de punto flotante **1.2345** en campos de varios tamaños. ¿Qué ocurre cuando los valores se imprimen en campos que contienen menos dígitos que los valores?

**21.10** Escriba un programa que imprima el valor **100.453627** redondeado al dígito más cercano, décimas, centésimas, milésimas y diezmilésimas.

**21.11** Escriba un programa que introduzca una cadena desde el teclado y que determine su longitud. Imprima la cadena utilizando el doble de la longitud como el ancho del campo.

**21.12** Escriba un programa que convierta temperaturas enteras en Fahrenheit desde **0** a **212** grados, a temperaturas Celsius en punto flotante con **3** dígitos de precisión. Utilice la fórmula

```
celsius = 5.0 / 9.0 * ( fahrenheit – 32 );
```

para realizar el cálculo. La salida debe imprimirse en dos columnas justificadas a la derecha, y las temperaturas Celsius deben estar precedidas por un signo, tanto para valores positivos como negativos.

**21.13** En algunos lenguajes de programación, las cadenas se introducen rodeadas por comillas sencillas o dobles. Escriba un programa que lea las tres cadenas **susy**, "**susy**" y '**susy**'. ¿Las comillas sencillas y dobles, se ignoran o se leen como parte de la cadena?

**21.14** En la figura 18.3, se sobrecargaron los operadores de extracción y de inserción de flujo para introducir y desplegar los objetos de la clase **NumeroTelefonico**. Rescriba el operador de extracción de flujo para realizar la siguiente verificación de errores en la entrada. La función **operator>>** deberá volverse a codificar completamente.
a) Introduzca el número telefónico completo en un arreglo. Verifique que se introdujo el número de caracteres correcto. Debe haber un total de 14 caracteres leídos para un número telefónico de la forma **(800) 555-1212**. Utilice la función miembro de flujo **clear** para establecer **ios::failbit** para entradas incorrectas.
b) El código de área e intercambio no comienzan con **0** o con **1**. Verifique que el primer dígito del código de área y las partes de intercambio del número telefónico no comiencen con **0** o **1**. Utilice la función miembro de flujo **clear** para establecer **ios::failbit** para entradas incorrectas.
c) El dígito de en medio de un código de área por lo general siempre es **0** o **1** (aunque recientemente esto ha cambiado). Verifique que el dígito central sea **0** o **1**. Utilice la función miembro **clear** para establecer **ios::failbit** para entradas incorrectas. Si ninguna de las operaciones anteriores resulta en un **ios::failbit**, comience con la configuración de entradas incorrectas, copie las tres partes del número telefónico en los miembros **codigoArea**, intercambio y **linea** del objeto **NumeroTelefonico**. En el programa

principal, si **ios::failbit** se estableció en la entrada, haga que el programa imprima un mensaje de error y que termine, en lugar de que imprima el número telefónico.

**21.15** Escriba un programa que realice las siguientes tareas:

a) Genere una clase definida por el usuario, **Punto**, que contenga los datos miembros privados enteros **coordenadaX** y **coordenadaY**, y que declare los operadores de inserción y de extracción de flujo sobrecargados como amigas de la clase.

b) Defina las funciones de operador de inserción y de extracción de flujo. La función de operador de extracción de flujo debe determinar si los datos introducidos son válidos, y si no es así, debe establecer **ios::failbit** para indicar una entrada incorrecta. El operador de inserción de flujo no debe poder desplegar el punto después de ocurrido un error de entrada.

c) Escriba una función **main** que evalúe la entrada y la salida de la clase Punto definida por el usuario, utilizando los operadores sobrecargados de inserción y extracción de flujo.

**21.16** Escriba un programa que realice cada una de las siguientes tareas:

a) Genere la clase **Complejo** definida por el usuario que contenga los datos miembro privados enteros real e imaginario, y declare a los operadores sobrecargados de inserción y de extracción de flujo como amigas de la clase.

b) Defina las funciones de operador de inserción y de extracción de flujo. El operador de extracción debe determinar si los datos introducidos son válidos, y si no es así, debe establecer **ios::failbit** para indicar una entrada incorrecta. La entrada debe ser de la forma

3 + 8i

c) Los valores pueden ser positivos o negativos, y es posible que uno de los dos valores no se proporcione. Si un valor no se proporciona, el dato miembro apropiado debe establecerse en 0. El operador de inserción de flujo no debe poder desplegar el punto, si ocurrió un error de entrada. El formato de salida debe ser idéntico al formato de entrada que mostramos arriba. Para valores imaginarios negativos debe imprimirse un signo menos, en lugar de un signo más.

d) Escriba una función **main** que evalúe la entrada y la salida de la clase **Complejo** definida por el usuario, utilizando los operadores de inserción y de extracción de flujo.

**21.17** Escriba un programa que utilice una estructura **for** para que imprima una tabla de valores ASCII que corresponda a los caracteres del conjunto ASCII del 33 al 126. El programa debe imprimir el valor decimal, el valor octal, el valor hexadecimal y el valor del carácter para cada carácter. Utilice los manipuladores de flujo **dec**, **oct** y **hex** para imprimir los valores enteros.

**21.18** Escriba un programa que muestre que las funciones miembro de **istream, getline** y **get** de tres argumentos finalizan la cadena de entrada con un carácter de terminación nulo. Además, que muestre que **get** deja al carácter delimitador en el flujo de entrada, mientras que **getline** lo extrae y lo descarta. ¿Qué ocurre con los caracteres no leídos del flujo?

**21.19** Escriba un programa que genere el manipulador **ignorablancos** definido por el usuario para que ignore los caracteres blancos a la izquierda del flujo de entrada. El manipulador debe utilizar la función **isspace** de la biblioteca <**cctype**>, para evaluar si el carácter es un blanco. Cada carácter debe introducirse por medio de la función miembro **get** de **istream**. Cuando se encuentra un carácter que no es blanco, el manipulador **ignorablancos** termina su trabajo colocando el carácter de regreso al flujo de entrada y devolviendo una referencia **istream**.

Evalúe el manipulador creando una función **main** en la que la bandera **ios::skipws** no esté establecida, para que el operador de extracción de flujo no ignore automáticamente los caracteres blancos. Después evalúe el manipulador en el flujo de entrada, introduciendo un carácter precedido por un carácter blanco como entrada. Imprima el carácter que se introdujo para confirmar que no se introdujo un carácter blanco.

# 22

# Plantillas en C++

## Objetivos

- Utilizar las plantillas de clases para crear un grupo de tipos relacionados.
- Diferenciar las plantillas de clases y las clases de plantillas.
- Comprender cómo sobrecargar plantillas de funciones.
- Comprender las relaciones entre plantillas, **`amigas`**, herencia y miembros **`estáticos`**.




*Detrás de ese patrón externo,*
*las tenues figuras se aclaran día con día.*
*Siempre es la misma figura, sólo que muy numerosa.*
Charlotte Perkins Gilman

*Si eres capaz de deslizarte a través de los cielos y la tierra,*
*entonces hazlo*.
El Corán

*¡Un extraordinario laberinto! Pero no sin un plano.*
Alexander Pope

## Plan general



## 22.1 Introducción

En este capítulo explicaremos una de las características más poderosas de C++, a saber, las plantillas. Las plantillas nos permiten especificar, con un solo segmento de código, un rango completo de funciones (sobrecargadas) relacionadas (llamadas funciones de plantilla) o un rango entero de clases relacionadas, llamadas *clases de plantillas*.

Como explicamos en el capítulo 15, podríamos escribir una *plantilla de función* individual para una función de ordenamiento de arreglos, y después hacer que C++ genere funciones de plantilla por separado que ordenen un arreglo **`int`**, un arreglo **`float`**, un arreglo de cadenas, y así sucesivamente.

Podríamos escribir una *plantilla de clases* individual para una clase de pila, y luego hacer que C++ genere clases de plantilla separadas tales como una clase de pila de enteros, una clase de pila de **`floats`**, una clase de pila de cadenas, y así sucesivamente.

Observe la diferencia entre plantillas de clases y clases de plantillas: las plantillas de clases son patrones a partir de los cuales trazamos figuras; las clases de plantillas son como trazos separados que tienen la misma forma pero se pueden dibujar, por ejemplo, con diferentes colores.



**Observación de ingeniería de software 22.1**

*Las plantillas son una de las capacidades más poderosas para la reutilización de software en C++.*

En este capítulo, presentaremos ejemplos de plantillas de clases. También consideraremos las relaciones entre las plantillas y otras características de C++, tales como la herencia, las amigas y los miembros estáticos.

El diseño y los detalles de los mecanismos de las plantillas que aquí explicamos se basan en el trabajo de Bjarne Stroustrup tal como lo presentó en su documento, *Parameterized Types for C++*, y publicado en *Proceedings of the USENIX C++ Conference* llevado a cabo en Denver, Colorado, en octubre de 1988.

## 22.2 Plantillas de clases

Es posible comprender qué es una pila (una estructura de datos en la que insertamos elementos en un orden y los recuperamos en el orden último en salir, primero en entrar) independientemente del tipo de elementos que se coloquen en ella. Pero cuando en realidad se trata de crear la instancia de una pila, debemos especificar un tipo de dato. Esto crea una maravillosa oportunidad para la reutilización de software. Necesitamos los medios para describir la noción de una pila de manera genérica y crear las instancias a partir de las clases, que son versiones específicas de esta clase genérica. En C++, esta capacidad la proporcionan las *plantillas de clases*.



**Observación de ingeniería de software 22.2**

*Las plantillas de clases promueven la reutilización de software, al permitir versiones para tipos específicos de las clases genéricas que van a instanciarse.*

A las plantillas de clases se les llama *tipos parametrizados*, debido a que requieren uno o más parámetros de tipo para especificar cómo personalizar una plantilla de "clase genérica" para formar una clase de plantilla específica.

El programador que desea producir una variedad de clases de plantillas simplemente escribe una definición de plantilla de clase. Cada vez que el programador necesita crear una nueva instancia de un tipo específico, utiliza una notación sencilla y concisa y el compilador escribe el código fuente para la clase de la plantilla que requiere el programador. Por ejemplo, una plantilla de clase **Pila** podría convertirse en la base para crear muchas clases **Pila** (tales como "**Pilas** de **double**s", "**Pilas** de **int**s", "**Pilas** de **char**s", "**Pilas** de **Empleados**", etcétera.), para utilizarlas dentro de un programa.

Observe la definición de la plantilla de clase **Pila** en la figura 22.1. Parece una definición tradicional de una clase, excepto por que va precedida por el encabezado (línea 6)

```
template< class T>
```

para especificar que es una definición de una plantilla de clase con el parámetro de tipo **T** que indica el tipo de la clase **Pila** a crearse. El programador no necesita utilizar **T** específicamente (es posible utilizar cualquier identificador). El tipo de elemento que va a almacenarse en esta **Pila** se menciona solamente de manera genérica como **T**, a través del encabezado de la clase **Pila** y de la definición de las funciones miembro. Por ahora mostraremos cómo es que **T** se asocia con un tipo específico, tal como un **double** o un **int**. Existen dos restricciones para los tipos de datos no primitivos que se utilizan en esta **Pila**: deben tener un constructor predeterminado y deben soportar el operador de asignación. Si un objeto de la clase que se utiliza en esta **Pila** contiene memoria asignada dinámicamente, debe sobrecargarse el operador de asignación para dicho tipo, como muestra el capítulo 18.

```
1  // Figura 22.1: tpila1.h
2  // Plantilla de clase Pila
3  #ifndef TPILA1_H
4  #define TPILA1_H
5
6  template< class T >
7  class Pila {
8  public:
9     Pila( int = 10 );    // constructor predeterminado (el tamaño de la
                                 pila es 10)
10    ~Pila() { delete [] ptrPila; } // destructor
11    bool push( const T& );  // coloca un elemento en la pila
12    bool pop( T& );         // saca un elemento de la pila
13 private:
14    int tamanio;              // # de elementos en la pila
15    int cima;                 // ubicación del elemento cima
16    T *ptrPila;               // apuntador a la pila
17
18    bool estaVacia() const { return cima == -1; }       // funciones de
19    bool estaLlena() const { return cima == tamanio - 1; } // utilidad
20 }; // fin de la plantilla de clase Pila
21
22 // Constructor con un tamaño predeterminado de 10
23 template< class T >
24 Pila< T >::Pila( int tam )
25 {
26    tamanio = tam > 0 ? tam : 10;
27    cima = -1;                  // La Pila inicialmente está vacía
28    ptrPila = new T[ tamanio ];  // asigna espacio para los elementos
29 } // fin del constructor Pila
30
31 // Coloca un elemento en la pila
32 // devuelve 1 si tiene éxito, de lo contrario devuelve 0
```

**Figura 22.1** Demostración de una plantilla de clase **Pila**; **tpila1.h**. (Parte 1 de 2.)

```
template< class T >
bool Pila< T >::push( const T &colocaValor )
{
   if ( !estaLlena() ) {
      ptrPila[ ++cima ] = colocaValor; // coloca el elemento en la Pila
      return true;  // si la colocación fue exitosa
   } // end if
   return false;     // si la colocación no fue exitosa
} // fin de la plantilla de función push

// Saca un elemento de la pila
template< class T >
bool Pila< T >::pop( T &sacaValor )
{
   if ( !estaVacia() ) {
      sacaValor = ptrPila[ cima-- ];  // saca el elemento de la Pila
      return true;  // si la eliminación fue exitosa
   } // end if
   return false;     // si la eliminación no fue exitosa
} // fin de la plantilla de función pop

#endif
```

**Figura 22.1** Demostración de una plantilla de clase **`Pila`**; **`tpila1.h`**. (Parte 2 de 2.)

```
// Figura 22.1: fig22_01.cpp
// Controlador de prueba para la plantilla Pila
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include “tpila1.h”

int main()
{
   Pila< double > pilaDouble( 5 );
   double d = 1.1;
   cout << “Colocando elementos en la pilaDouble\n”;

   while ( pilaDouble.push( d ) ) { // éxito, true devuelto
      cout << d << ‘ ‘;
      d += 1.1;
   } // fin de while

   cout << “\nLa pila esta llena. No se puede colocar “ << d
        << “\n\nSacando elementos de la pilaDouble\n”;

   while ( pilaDouble.pop( d ) )  // éxito, true devuelto
      cout << d << ‘ ‘;

   cout << “\nLa pila esta vacia. No se puede sacar un elemento\n”;

```

**Figura 22.1** Demostración de una plantilla de clase **`Pila`**; **`fig22_01.cpp`**. (Parte 1 de 2.)

```
84      Pila< int > pilaInt;
85      int i = 1;
86      cout << “\nColocando elementos en la pilaInt\n”;
87
88      while ( pilaInt.push( i ) ) { // éxito, true devuelto
89         cout << i << ‘ ‘;
90         ++i;
91      } // fin de while
92
93      cout << “\nLa pila esta llena. No se puede colocar “ << i
94           << “\n\nSacando elementos de la pilaInt\n”;
95
96      while ( pilaInt.pop( i ) )  // éxito, true devuelto
97         cout << i << ‘ ‘;
98
99      cout << “\nLa pila esta vacia. No se puede sacar un elemento\n”;
100     return 0;
101  } // fin de la función main
```

```
Colocando elementos en la pilaDouble
1.1 2.2 3.3 4.4 5.5
La pila esta llena. No se puede colocar 6.6

Sacando elementos de la pilaDouble
5.5 4.4 3.3 2.2 1.1
La pila esta vacia. No se puede sacar un elemento

Colocando elementos en la pilaInt
1 2 3 4 5 6 7 8 9 10
La pila esta llena. No se puede colocar 11

Sacando elementos de la pilaInt
10 9 8 7 6 5 4 3 2 1
La pila esta vacia. No se puede sacar un elemento
```

**Figura 22.1** Demostración de una plantilla de clase **Pila**; **fig22_01.cpp**. (Parte 2 de 2.)

Ahora consideremos el controlador (**main**) que ejecuta la plantilla de clase **Pila** (vea la salida en la figura 22.1). El controlador comienza con la creación de la instancia del objeto **pilaDouble** de tamaño **5**. Este objeto se declara de clase **Pila< double >** (que se pronuncia “**Pila** de **double**s”). El compilador asocia el tipo **double** con el parámetro de tipo **T** en la plantilla para producir el código fuente para la clase **Pila** de tipo **double**. Aunque el programador no ve este código fuente, sí se incluye en el código fuente y se compila.

Después, el controlador coloca sucesivamente los valores de tipo **double** 1.1, 2.2, 3.3, 4.4 y 5.5 dentro de **pilaDouble**. El ciclo **push** termina cuando el controlador intenta colocar un sexto valor dentro de **pilaDouble** (la cual ya está llena debido a que fue creada para almacenar un máximo de cinco elementos).

Ahora, el controlador saca los cinco valores de la pila (observe en la figura 22.1 que los valores se sacan en el orden último en entrar, primero en salir). El controlador intenta sacar un sexto valor, pero **pilaDoubles** ya está vacía, de modo que el ciclo termina.

A continuación, el controlador crea la instancia de **pilaInt** con la declaración

```
Pila< int > pilaInt;
```

(que se lee: “**pilaInt** es una **Pila** de **int**s”). No se especifica tamaño, de manera que se establece el tamaño predeterminado de 10 dentro del constructor predeterminado (línea 24). Una vez más, el controlador hace

el ciclo y coloca los valores dentro de **pilaInts** hasta llenarla, después, hace el ciclo y saca los valores de **pilaInt** hasta que se vacía. Una vez más, los valores se sacan en orden último en entrar, primero en salir.

Cada definición de las funciones miembro fuera de la clase comienza con el encabezado (línea 23)

```
template< class T >
```

Entonces cada definición parece una definición tradicional de función, excepto que el tipo del elemento **Pila** por lo general se lista como un parámetro de tipo **T**. El operador binario de resolución de alcance se utiliza con el nombre de la plantilla de clase **Pila< T >** para relacionar cada definición de función miembro con el alcance de la plantilla de clase. En este caso, el nombre de la clase es **Pila< T >**. Cuando se crea la instancia **pilaDouble** para que sea del tipo **Pila< double >**, el constructor de **Pila** utiliza **new** para crear un arreglo de elementos de tipo **double** que represente a la pila (línea 28). La instrucción

```
ptrPila = new T[ tamanio ];
```

de la definición de la plantilla de clase **Pila** es generada por el compilador en la clase de plantilla **Pila< double** > como

```
ptrPila = new double[ tamanio ];
```

Observe que el código en la función **main** de la figura 22.1 es casi idéntica para ambas manipulaciones de **pilaDoubles** en la mitad superior de **main** y en las manipulaciones de **pilaInt** en la mitad inferior de **main**. Esto nos presenta otra oportunidad para utilizar una plantilla de función. La figura 22.2 utiliza la plantilla de función **pruebaPila** para realizar las mismas tareas que **main** en la figura 22.1; coloca una serie de valores dentro de **Pila< T >** y saca los valores de **Pila< T >**. La plantilla de función **pruebaPila** utiliza un parámetro de tipo formal **T** para representar el tipo de dato almacenado en **Pila< T >**. La plantilla de función toma cuatro argumentos, una referencia a un objeto de tipo **Pila< T >**, un valor de tipo **T** que será el primer valor colocado dentro de **Pila< T >**, un valor de tipo **T** que se utiliza para incrementar los valores colocados dentro de **Pila< T >** y una cadena de caracteres de tipo **const char *** que representa el nombre del objeto para propósitos de salida. Ahora, la función **main** simplemente crea la instancia de un objeto de tipo **Pila< double >** llamado **pilaDouble** y un objeto de tipo **Pila< int >** llamado **pilaInt** y utiliza estos objetos en las líneas 42 y 43.

```
pruebaPila( pilaDouble, 1.1, 1.1, “pilaDouble” );
pruebaPila( pilaInt, 1, 1, “pilaInt” );
```

Observe que la salida de la figura 22.2 coincide de manera precisa con la salida de la figura 22.1.

## 22.3 Plantillas de clases y parámetros sin tipo

La plantilla de clase **Pila** de la sección anterior utilizaba solamente parámetros de tipo dentro del encabezado de la plantilla. También es posible utilizar *parámetros sin tipo*; un parámetro sin tipo puede tener un argumento predeterminado, y puede tratarse como **const**. Por ejemplo, el encabezado de la plantilla puede modificarse para tomar un parámetro **int elementos** de la siguiente manera:

```
template< class T >, int elementos; // observe el parámetro sin tipo
```

Después, una declaración como

```
Pila< double, 100 > cifrasDeVentasMasRecientes;
```

creará la instancia (en tiempo de compilación) de una clase de plantilla con 100 elementos de nombre **cifrasDeVentasMasRecientes** con valores **double**; esta clase de plantilla sería de tipo **Pila< double, 100 >**. El encabezado de la clase podría contener un dato miembro privado con una declaración de arreglo como

```
T contenedorPila[ elementos ];   // arreglo para almacenar el contenido
                                    de la Pila
```

```
// Figura 22.2: fig22_02.cpp
// Controlador de prueba para la plantilla Pila.
// La función main utiliza una plantilla de función para manipular
// objetos del tipo Pila< T >.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include “tpila1.h”

// Plantilla de función para manipular Pila< T >
template< class T >
void pruebaPila(
   Pila< T > &laPila,       // referencia hacia la Pila< T >
   T valor,                 // valor inicial a colocarse
   T incremento,            // incremento para valores subsiguientes
   const char *nombrePila ) // nombre del objeto Pila < T >
{
   cout << “\nColocando elementos en “ << nombrePila << ‘\n’;

   while ( laPila.push( valor ) ) { // éxito, true devuelto
      cout << valor << ‘ ‘;
      valor += incremento;
   } // end while

   cout << “\nLa pila esta llena. No se puede colocar otro elemento “ << valor
        << “\n\nSacando elementos de “ << nombrePila << ‘\n’;

   while ( laPila.pop( valor ) )  // éxito, true devuelto
      cout << valor << ‘ ‘;

   cout << “\nLa pila esta vacia. No se puede sacar un elemento\n”;
} // fin de la plantilla de función pruebaPila

int main()
{
   Pila< double > pilaDouble( 5 );
   Pila< int > pilaInt;

   pruebaPila( pilaDouble, 1.1, 1.1, “pilaDouble” );
   pruebaPila( pilaInt, 1, 1, “pilaInt” );

   return 0;
} // fin de la función main
```

```
Colocando elementos en pilaDouble
1.1 2.2 3.3 4.4 5.5
La pila esta llena. No se puede colocar 6.6

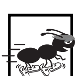
Sacando elementos de pilaDouble
5.5 4.4 3.3 2.2 1.1
La pila esta vacia. No se puede sacar un elemento
```

**Figura 22.2** Paso de un objeto de la plantilla **`Pila`** a una plantilla de función. (Parte 1 de 2.)

```
Colocando elementos en pilaInt
1 2 3 4 5 6 7 8 9 10
La pila esta llena. No se puede colocar 11

Sacando elementos de pilaInt
10 9 8 7 6 5 4 3 2 1
La pila esta vacia. No se puede sacar un elemento
```

**Figura 22.2** Paso de un objeto de la plantilla **Pila** a una plantilla de función. (Parte 2 de 2.)



**Tip de rendimiento 22.1**

*Cuando es posible hacerlo, especificar el tamaño de una clase contenedora (tal como una clase arreglo o una clase pila) en tiempo de compilación (posiblemente a través de un parámetro de tamaño de una plantilla sin tipo), elimina el exceso de tiempo de ejecución correspondiente a la creación dinámica de espacio por medio de* **`new`**.



**Observación de ingeniería de software 22.3**

*Cuando es posible hacerlo, especificar el tamaño de una clase contenedora en tiempo de compilación (posiblemente a través de un parámetro de tamaño de una plantilla sin tipo) elimina la posibilidad de un error fatal en tiempo de ejecución, si* **`new`** *es incapaz de obtener la memoria necesaria.*

En los ejercicios, usted utilizará un parámetro sin tipo para crear una plantilla para la clase **Arreglo** que desarrollamos en el capítulo 18. Esta plantilla permite la creación de las instancias de los objetos **Arreglo** con un número específico de elementos de un tipo específico en tiempo de compilación, en lugar de crear de manera dinámica el espacio para los objetos **Arreglo** en tiempo de ejecución.

Una clase para un tipo específico que no coincide con una plantilla de clase común puede proporcionarse para redefinir la plantilla de clase para ese tipo. Por ejemplo, puede utilizarse una plantilla de la clase **Arreglo** para instanciar un arreglo de cualquier tipo. El programador puede elegir tomar el control de la creación de la instancia de la clase **Arreglo** de un tipo específico, tal como **Marciano**. Esto se hace simplemente al formar la nueva clase con un nombre de la clase **Arreglo< Marciano >**.

## 22.4 Plantillas y herencia

Las plantillas y la herencia se relacionan de distintas maneras:

- Una plantilla de clase puede derivarse a partir de una clase de plantilla.
- Una plantilla de clase puede derivarse a partir de una clase que no es plantilla.
- Una plantilla de clase puede derivarse a partir de una plantilla de clase.
- Una clase que no es plantilla puede derivarse a partir de una plantilla de clase.

## 22.5 Plantillas y amigas

Hemos visto que las funciones y las clases completas pueden declararse como amigas de clases que no son plantillas. Con las plantillas de clases, pueden declararse los tipos obvios de arreglos de amistad. La amistad puede establecerse entre una plantilla de clase y una función global, una función miembro de otra clase (posiblemente una clase de plantilla), o incluso una clase completa (posiblemente una clase de plantilla). Las notaciones que se requieren para establecer estas relaciones de amistad pueden ser engorrosas.

Dentro de una plantilla de clase correspondiente a la clase **X** que se declaró con

```
template< class T > class X
```

una declaración de amistad de la forma

```
friend void f1();
```

hace de la función **f1** una amiga de cada clase de plantilla instanciada a partir de la plantilla de clase anterior.

Dentro de una plantilla de clase correspondiente a la clase **X** que se declaró como

```
template< class T > class X
```

una declaración de amistad de la forma

```
friend void f2( X< T > & );
```

para un tipo particular **T** tal como **float**, hace de la función **f2( X< float>& )** una amiga sólo de **X< float** >.

Dentro de una plantilla de clase, usted puede declarar que una función miembro de otra clase sea una amiga de cualquier clase generada a partir de la plantilla de clase. Simplemente nombre a la función miembro de otra clase mediante el nombre de la clase y el operador binario de resolución de alcance. Por ejemplo, dentro de la plantilla de clase correspondiente a la clase **X** que se declaró con

```
template< class T > class X
```

una declaración de amistad de la forma

```
friend void A::f4();
```

hace de la función miembro **f4** de la clase **A** una amiga de cada clase de plantilla instanciada a partir de la plantilla de clase anterior

Dentro de una plantilla de clase correspondiente a la clase **X** que se declaró con

```
template< class T > class X
```

una declaración de amistad de la forma

```
friend void C< T >::f5( X< T > & );
```

para un tipo particular **T** tal como **float**, hace de la función miembro

```
C< float >::f5( X< float > & )
```

una función amiga *solamente* de la clase **X< float >**.

Dentro de una plantilla de clase correspondiente a la clase **X** que se declaró con

```
template< class T > class X
```

puede declararse una segunda clase **Y** con

```
friend class Y;
```

lo que hace de cada función miembro de la clase **Y** una amiga de cada clase de plantilla producida a partir de la plantilla de clase **X**.

Dentro de una plantilla de clase correspondiente a la clase **X** que se declaró con

```
template< class T > class X
```

puede declararse una segunda clase **Z** con

```
friend class Z< T > ;
```

entonces, cuando se crea la instancia de una clase de plantilla con el tipo particular para **T** tal como **float**, todos los miembros de **class Z< float >** se vuelven amigas de la clase de plantilla **X< float >**.

## 22.6 Plantillas y miembros estáticos

¿Y qué sucede con los datos miembro estáticos? Recuerde que en una clase que no es plantilla, se comparte una copia del dato miembro estático entre todos los objetos de la clase, y que los datos miembro estáticos deben declararse con alcance de archivo.

Cada clase de plantilla instanciada a partir de una plantilla de clase contiene su propia copia de cada dato miembro estático de la plantilla de clase; todos los objetos de dicha clase de plantilla comparten dicho dato miembro estático. Y como sucede con los datos miembro no estáticos de las clases que no son plantillas, los datos miembros estáticos de las clases de plantillas deben inicializarse con alcance de archivo. Cada clase de plantilla obtiene su propia copia de la plantilla de las funciones miembro estáticas de la plantilla de clase.

## RESUMEN

- Las plantillas nos permiten especificar un rango de funciones relacionadas (sobrecargadas), llamadas funciones de plantillas, o un rango de clases relacionadas, llamadas plantillas de clases.
- Las plantillas de clase proporcionan los medios para describir una clase de manera genérica y para crear instancias de las clases que son de tipo específico para esta clase genérica.
- A las plantillas de clases se les llama tipos parametrizados; éstos requieren parámetros de tipo para especificar cómo personalizar un plantilla de clase genérica para formar una plantilla de clase específica.
- El programador que desee utilizar clases de plantillas escribe una plantilla de clase. Cuando un programador necesita un nuevo tipo específico de clase, utiliza una notación concisa y el compilador escribe el código fuente para la plantilla de la clase de la plantilla.
- Una definición de una plantilla de clase se parece a una definición tradicional de una clase, excepto que la primera va precedida por **`template< class T>`** (o **`template< nombreTipo T >`**) para indicar que es una definición de una plantilla de clase, en donde el parámetro **`T`** indica el tipo de la clase que se va a crear. El tipo **`T`** se menciona a través del encabezado de la clase y de la definición de las funciones miembro como un nombre genérico de tipo.
- Las definiciones de las funciones miembro fuera de la clase comienzan con el encabezado **`template< class T>`** (o **`Template< nombreTipo T>`**). Entonces, cada definición de función nos recuerda a la definición de una función miembro, con la excepción de que los datos genéricos de la clase siempre se listan de manera general como parámetros de tipo **`T`**. El operador binario de resolución de alcance se utiliza con el nombre de la plantilla de la clase para relacionar cada definición de función miembro con el alcance de la plantilla de la clase, como en **`NombreClase<T>`**.
- Es posible utilizar parámetros sin tipo en el encabezado de la plantilla de la clase.
- Es posible proporcionar una clase para un tipo específico para redefinir la plantilla de la clase para ese tipo.
- A partir de una clase de plantilla puede derivarse una plantilla de clase. Una clase de plantilla puede derivarse a partir de una clase que no es plantilla. Una clase de plantilla puede derivarse a partir de una plantilla de clase. Una clase que no es plantilla puede derivarse a partir de una plantilla de clase.
- Las funciones y todas las clases pueden declararse como amigas de las clases que no son plantillas. Con las plantillas de clases, pueden declararse los tipos obvios de arreglos de amistad. La amistad puede establecerse entre una plantilla de clase y una función global, una función miembro de otra clase (posiblemente una clase de plantilla) o incluso una clase completa (posiblemente una clase de plantilla).
- Cada clase de plantilla instanciada a partir de una plantilla de clase contiene su propia copia de cada dato miembro estático de la plantilla de clase; todos los objetos de esa clase de plantilla comparten ese dato miembro estático. Y así como sucede con los datos miembro estáticos para las clases que no son plantillas, los datos miembro estáticos de las clases de plantillas deben inicializarse con alcance de archivo.
- Cada clase de plantilla obtiene una copia de las funciones miembro estáticas de la plantilla de clase.

## TERMINOLOGÍA

amiga de una plantilla
argumento de plantilla
clase de plantilla
dato miembro estático de una clase de plantilla
dato miembro estático de una plantilla de clase
función de plantilla
función miembro de la clase de plantilla
función miembro estática de una clase de plantilla
función miembro estática de una plantilla de clase
nombre de plantilla
nombre de una plantilla de clase
palabra reservada **`class`** en un parámetro de tipo en la plantilla
palabra reservada **`template`**
parámetro de plantilla
parámetro de tipo en un encabezado de plantilla
parámetro de tipo formal en el encabezado de una plantilla
parámetro sin tipo en un encabezado de plantilla
plantilla de clase
sobrecarga de una función de plantilla
**`template< class T >`**
tipo parametrizado
**`typename`**

## TIP DE RENDIMIENTO

**22.1** Cuando es posible hacerlo, especificar el tamaño de una clase contenedora (tal como una clase arreglo o una clase pila) en tiempo de compilación (posiblemente a través de un parámetro de tamaño de una plantilla sin tipo), elimina el exceso de tiempo de ejecución correspondiente a la creación dinámica de espacio por medio de **`new`**.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**22.1** Las plantillas son una de las capacidades más poderosas para la reutilización de software en C++.

**22.2** Las plantillas de clases promueven la reutilización de software, al permitir versiones para tipos específicos de las clases genéricas que van a instanciarse.

**22.3** Cuando es posible hacerlo, especificar el tamaño de una clase contenedora en tiempo de compilación (posiblemente a través de un parámetro de tamaño de una plantilla sin tipo) elimina la posibilidad de un error fatal en tiempo de ejecución, si **new** es incapaz de obtener la memoria necesaria.

## EJERCICIOS DE AUTOEVALUACIÓN

**22.1** Conteste *verdadero* o *falso*. Si su respuesta es *falso*, explique por qué.

a) Si se generan varias clases de plantillas desde una sola plantilla de clase con un solo dato miembro, cada una de las clases de la plantilla comparte una copia individual del dato miembro estático de la plantilla de la clase.
b) El nombre de un parámetro de tipo formal solamente puede utilizarse una vez en la lista de parámetros de tipo formal de la definición de la plantilla. Los nombres de los parámetros de tipo formal deben ser únicos a lo largo de las definiciones de la plantilla.
c) Las palabras reservadas **class** y **typename**, tal como se utilizan con un parámetro de tipo de la platilla significan específicamente "cualquier tipo de clase definida por el usuario".

**22.2** Complete los espacios en blanco:

a) Las plantillas nos permiten especificar, mediante un solo segmento de código, un rango completo de clases relacionadas llamadas ____________.
b) A las plantillas de clases también se les llama tipos ____________.
c) El operador ____________ se utiliza con un nombre de clase de plantilla para relacionar cada definición de función miembro con el alcance de la plantilla de la clase.
d) Así como con los datos miembro estáticos de una clase que no es plantilla, los datos miembro estáticos de las clases de plantillas también se deben inicializar con alcance de ____________.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**22.1** a) Falso. Cada clase de plantilla tendrá su propia copia del dato miembro estático. b) Falso. Los nombres de los parámetros de tipo formal necesitan ser únicos a lo largo de las funciones de la plantilla. c) Falso. Las palabras reservadas **class** y **typename** en este contexto también permiten un tipo de parámetro de tipo predeterminado.

**22.2** a) Clases de plantillas. b) Parametrizados. c) Binario de resolución de alcance. d) Archivo.

## EJERCICIOS

**22.3** Utilice el parámetro sin tipo **numeroDeElementos** y un parámetro de tipo **tipoElemento** para ayudar a crear una plantilla para la clase **Arreglo** que desarrollamos en el capítulo 18. Esta plantilla permitirá crear, en tiempo de compilación, las instancias de los objetos **Arreglo** con un número específico de elementos con el tipo de elemento específico.

**22.4** Escriba un programa con la plantilla de la clase **Arreglo**. La plantilla puede crear la instancia de un **Arreglo** de cualquier tipo de elementos. Ignore la plantilla con una definición específica para un **Arreglo** de elementos de tipo **float** (**class Arreglo<float>**). El controlador debe demostrar la creación de la instancia de un **Arreglo** de enteros a través de la plantilla, y debe mostrar que al intentar crear la instancia de un **Arreglo** de tipo **float** utiliza la definición proporcionada en **class Arreglo< float >**.

**22.5** ¿Qué se parece más a un patrón, una plantilla de clase o una clase de plantilla? Explique su respuesta.

**22.6** ¿Qué problema de rendimiento puede provocar el uso de plantillas de clases?

**22.7** ¿Por qué es apropiado llamar a una plantilla de clase tipo parametrizado?

**22.8** Explique por qué usted podría utilizar la instrucción

```
Arreglo< Empleado > listaEmpleado( 100 );
```

en un programa en C++.

**22.9** Revise su respuesta del ejercicio 22.8. Ahora, por qué podría utilizar la instrucción

```
Arreglo< Empleado > listaEmpleado;
```

en un programa en C++?

**22.10** Explique el uso de la siguiente notación dentro de un programa en C++.

```
template< class T > Arreglo< T >::Arreglo( int s )
```

**22.11** ¿Por qué utilizaría, por lo general, un parámetro sin tipo con una plantilla de clase para un contenedor, como una arreglo o una pila?

**22.12** Describa cómo proporcionar una clase de un tipo específico para ignorar la plantilla de clase para dicho tipo.

**22.13** Describa la relación entre plantillas de clases y la herencia.

**22.14** Suponga que una plantilla de clase tiene un encabezado

```
template< class T > class C1
```

Describa las relaciones de amistad establecidas al colocar cada una de las siguientes declaraciones de amistad, dentro de este encabezado de plantilla de clase. Los identificadores que comienzan con "**f**" sin funciones, los identificadores que comienzan con "**C**" son clases y los identificadores que comienzan con "**T**" pueden representar a cualquier tipo (es decir, tipos predeterminados o tipos de clases).

a) `friend void f1();`
b) `friend void f2( C1 < T1 > & );`
c) `friend void C2::f4();`
d) `friend void C3< T1 >::f5( C1 < T1 > & );`
e) `friend class C5;`
f) `friend class C6< T1 >;`

**22.15** Suponga que la plantilla de clase **Empleado** tiene el dato miembro estático **cuenta**. Suponga que estas clases de plantilla se instancian desde la plantilla de clase. ¿Cuántas copias del dato miembro estático existen? ¿Cómo se restringirá el uso de cada una (si existe alguna restricción)?

# 23

# Manejo de excepciones en C++




## Objetivos

- Utilizar **`try`**, **`throw`** y **`catch`** para prevenir, indicar y manipular excepciones, respectivamente.
- Procesar excepciones no atrapadas e inesperadas.
- Procesar fallas de **`new`**.
- Utilizar **`auto_ptr`** para prevenir fugas de memoria.
- Comprender la jerarquía estándar de las excepciones.

*Nunca olvido una cara, pero en tu caso haré una excepción.*
Groucho (Julio Enrique) Marx

*Ninguna regla es tan general, para no admitir excepciones.*
Robert Burton

*Es cuestión de sentido común adoptar un método y seguirlo.*
*Si falla, admítalo francamente e intente otro. Pero sobretodo,*
*intente algo.*
Franklin Delano Roosevelt

*¡Oh! elimina la peor parte de eso,*
*y vive lo más puro de la otra mitad.*
William Shakespeare

*Si corren y no ven hacia adónde van,*
*tengo que salir de alguna parte y atraparlos.*
Jerome David Salinger

*Excusarse con frecuencia por una falta,*
*hace que la falta sea peor por la excusa.*
William Shakespeare

*Errar es de humanos, el perdonar es divino.*
El Papa Alejandro

## Plan general



## 23.1 Introducción

En este capítulo, presentaremos el *manejo de excepciones*. La extensibilidad de C++ puede incrementar de manera importante el número y las clases de errores que pueden ocurrir. Las características que presentamos aquí permiten a los programadores escribir programas más claros, más robustos, y más tolerantes a fallas. Los sistemas recientes desarrollados con éstas y otras técnicas similares han reportado resultados positivos. Además mencionaremos cuándo debe evitarse el manejo de excepciones.

El estilo y los detalles del manejo de excepciones que presentamos en este capítulo se basan en el trabajo de Andrew Koenig y Bjarne Stroustrup presentado en el artículo, "Exception Handling for C++ (revised)", publicado en *Proceedings of the USENIX C++ Conference* la cual se llevó a cabo en San Francisco en abril de 1990.

El código para manejo de errores varía en naturaleza y cantidad a lo largo de los sistemas de software dependiendo de la aplicación y si es un producto o no para liberarse. Los productos comerciales tienden a contener mucho más código de manejo de errores que el software "informal".

Existen muchos medios populares para lidiar con los errores. Por lo general, el código para manejo de errores está intercalado a lo largo del código del sistema. Los errores se manejan en los lugares en donde ocurren. La ventaja de este método es que el programador que lee el código puede ver el procesamiento de errores en la vecindad inmediata del código, y determinar si se implementó la verificación de errores apropiada.

El problema con este esquema es que, de alguna manera, el código se "contamina" con el procesamiento de errores. A un programador preocupado por el propio código se le hace más difícil leerlo y determinar si éste funciona correctamente. Esto dificulta la comprensión y el dar mantenimiento al código.

Algunos ejemplos comunes de excepciones son las fallas de **`new`** al intentar obtener la cantidad solicitada de memoria, un subíndice de arreglo fuera de límite, un desbordamiento aritmético, una división entre cero y parámetros de función inválidos.

Las características de manejo de excepciones de C++ permiten al programador eliminar el código de manejo de errores de la "línea principal" de ejecución del programa. Esto mejora la claridad y la posibilidad de modificación del programa. Con el estilo de C++ para el manejo de excepciones, es posible atrapar todo tipo

de excepciones, atrapar todas las excepciones de cierto tipo, o atrapar todas las excepciones de tipos relacionados. Esto hace a los programas más robustos, al reducir la probabilidad de que los errores no sean atrapados por el programa. El manejo de excepciones se proporciona para permitir a los programadores atrapar y manipular errores, en lugar de permitir que ocurran y tener que sufrir las consecuencias. Si el programador no proporciona los medios para manejar los errores fatales, el programa terminará.

El manejo de errores está diseñado para lidiar con *errores síncronos*, tales como intentar realizar una división entre cero (que ocurre cuando el programa ejecuta la instrucción de división). Con el manejo de excepciones, antes de que el programa ejecute la división, verifica el denominador y "arroja" (lanza) una excepción si el denominador es cero.

El manejo de excepciones no está diseñado para lidiar con situaciones asíncronas tales como operaciones de E/S, mensajes de red, clics del ratón y otras similares; éstos se manejan mejor a través de otros medios, tales como el procesamiento de interrupciones.

El manejo de excepciones se utiliza en situaciones en las que el sistema puede recuperarse del error que provoca la excepción. Al procedimiento de recuperación se le llama *manipulador de eventos*. Por lo general, el manejo de excepciones se utiliza cuando el error se manejará mediante una parte diferente del programa (es decir, un alcance diferente), a partir del cual se detectó el error. Un programa que contiene un diálogo interactivo con el usuario no debe utilizar excepciones para procesar los errores de entrada.

El manejo de excepciones es especialmente apropiado en situaciones en las que el programa no es capaz de recuperarse, pero necesita establecer una limpieza ordenada, y posteriormente salir "con gracia".



**Buena práctica de programación 23.1**

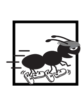
*Utilice excepciones para errores que deben procesarse en un alcance diferente al que ocurren. Utilice otros medios para manejar los errores que se procesarán en el mismo alcance en el que ocurren.*



**Buena práctica de programación 23.2**

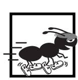
*Evite utilizar la manipulación de excepciones para otros propósitos que no sean la manipulación de errores, ya que puede reducir la claridad del programa.*

Existe otra razón para evitar el uso de técnicas de manipulación de excepciones para el control tradicional del programa. La manipulación de excepciones está diseñada para el procesamiento de errores, lo cual es una actividad poco frecuente que se utiliza generalmente debido a que el programa está a punto de terminar. Dada esta situación, no es necesario que los creadores de compiladores de C++ implementen la manipulación de excepciones para un óptimo rendimiento que podría esperarse en el código de aplicaciones normales.



**Tip de rendimiento 23.1**

*Aunque es posible utilizar la manipulación de excepciones para propósitos diferentes a la manipulación de errores, esto puede reducir el rendimiento del programa.*



**Tip de rendimiento 23.2**

*Por lo general, la manipulación de excepciones se implementa en los compiladores de tal manera que cuando no ocurre una excepción, existe poca o ninguna sobrecarga por la presencia de código de manipulación de excepciones. Cuando ocurren las excepciones, ocurre una sobrecarga en tiempo de ejecución. En realidad, la presencia de código para manipulación de excepciones hace que el programa consuma más memoria.*



**Observación de ingeniería de software 23.1**

*Por lo general, el flujo de control con estructuras de control tradicionales es más claro y más eficiente que con excepciones.*



**Error común de programación 23.1**

*Otra razón por la que las excepciones pueden ser peligrosas como una alternativa al flujo de control normal es que la pila se desenrolla y los recursos alojados antes de la ocurrencia de la excepción podrían no estar libres. Este problema puede evitarse por medio de una programación cuidadosa.*

La manipulación de excepciones ayuda a mejorar la tolerancia a fallas de los programas. Debido a que se vuelve más "placentero" escribir código para procesamiento de errores, es más probable que los programadores lo proporcionen. También es posible atrapar excepciones de otras formas, tales como por tipo, o incluso especificar el tipo de las excepciones a capturar.

La mayoría de los programas escritos en la actualidad soportan solamente un proceso en ejecución. El subprocesamiento múltiple recibe gran atención en los sistemas operativos actuales como Windows NT, OS/2 y distintas versiones de UNIX. Las técnicas que explicamos en éste capítulo se aplican incluso para programas de subprocesamiento múltiple, aunque no explicaremos específicamente programas con subprocesamiento múltiple.

Explicaremos cómo lidiar con excepciones "no atrapadas". Explicaremos cómo se manipulan las excepciones inesperadas. Mostraremos cómo pueden representarse las excepciones relacionadas por medio de clases de excepciones derivadas a partir de una clase de excepción de base común.

Las características de manipulación de excepciones en C++ se están utilizando ampliamente como resultado del estándar de C++. La estandarización es especialmente importante en proyectos grandes de software, en donde docenas o incluso cientos de personas trabajan en componentes separados de un sistema y estos componentes necesitan interactuar con el sistema completo para trabajar apropiadamente.



**Observación de ingeniería de software 23.2**

*El manejo de excepciones se adapta bien en sistemas con componentes desarrollados por separado. El manejo de excepciones facilita la combinación de componentes. Cada componente puede realizar su propia detección de excepciones, de manera separada de la manipulación de excepciones con otro alcance.*

La manipulación de excepciones puede considerarse como otro medio para devolver el control desde una función o desde la salida de un bloque de código. Por lo general, cuando ocurre una excepción, ésta será manipulada por la llamada a la función que genera la excepción, por una llamada a dicha llamada o por cualquier llamada en la cadena de llamadas para encontrar un manipulador para dicha excepción.

## 23.2 Cuándo debe utilizarse el manejo de excepciones

La manipulación de excepciones debe utilizarse para procesar solamente situaciones excepcionales, no obstante el hecho de que no existe manera alguna de prevenir a un programador sobre el uso de las excepciones como una alternativa del control del programa; para procesar las excepciones de los componentes de un programa que no están preparados para manipular dichas excepciones directamente; para procesar las excepciones de los componentes de software, tales como las funciones, las bibliotecas y las clases que probablemente tengan un uso constante, y en donde no tiene sentido que dichos componentes manipulen sus propias excepciones; y en grandes proyectos para manipular el procesamiento de errores de una manera uniforme a lo largo del proyecto.

**Buena práctica de programación 23.3**

*Utilice técnicas tradicionales de manejo de errores en lugar de la manipulación de excepciones, para procesar errores locales de manera directa, en donde sea fácil para un programa lidiar con sus propios errores.*



**Observación de ingeniería de software 23.3**

*Cuando trabaje con bibliotecas, es probable que quien llama a la función de la biblioteca tendrá en mente un procesamiento de errores único para una excepción que se genera en la función de la biblioteca. Es poco probable que una función de biblioteca realice el procesamiento de errores que coincida con las necesidades únicas de todos los usuarios. Por lo tanto, las excepciones son un medio apropiado para lidiar con los errores producidos por las funciones de bibliotecas.*

## 23.3 Otras técnicas de manejo de errores

Antes del presente capítulo, explicamos una variedad de formas para lidiar con situaciones excepcionales. Los siguientes puntos resumen éstas y otras técnicas útiles:

- Utilice **`assert`** para evaluar errores de código y de diseño. Si una afirmación es falsa, el programa termina y el código debe corregirse. Esto es útil en tiempo de depuración.
- Simplemente ignore las excepciones. Esto sería devastador para los productos de software liberados para el público en general, o para software de propósito especial necesario para situaciones de misión crítica. Pero para su propio software y para sus propios propósitos, es muy común ignorar muchos tipos de errores.
- Abandone el programa. Por supuesto, esto evita que un programa se ejecute completamente y que produzca resultados incorrectos. En realidad, esto es apropiado para muchos tipos de errores, en especial

para errores no fatales que permiten a un programa ejecutarse por completo, quizá engañando al programador para que piense que se ejecutó de manera correcta. Aquí, dicha estrategia también es inapropiada para aplicaciones de misión crítica. Los temas respecto a los recursos también son importantes aquí. Si un programa obtiene un recurso, por lo general, el programa debe devolver dicho recurso antes de que el programa termine.



**Error común de programación 23.2**

*Abandonar un programa puede dejar a un recurso en un estado en el que los demás programas no podrán adquirir dicho recurso; por lo tanto, el programa tendrá una "fuga de recursos".*

- Establezca algún indicador de error. El problema con esto es que es probable que los programas no verifiquen estos indicadores de error en todos los puntos en los que los errores pueden ser problemáticos.
- Verifique la condición del error, lance un mensaje de error y una llamada a **`exit`** para pasar un código de error apropiado al entorno del programa.
- **`setjump`** y **`longjump`**. Estas funciones de la biblioteca **`<csetjmp>`** permiten al programador especificar un salto inmediato fuera de las llamadas a funciones profundamente anidadas hacia un manipulador de error. Sin **`setjump/longjump`**, un programa debe ejecutar varias instrucciones **`return`** para salir de las llamadas a funciones anidadas. Estas funciones podrían utilizarse para saltar hacia un manipulador de error, pero son peligrosas debido a que desenrollan la pila sin llamar a los destructores de los objetos automáticos. Esto puede provocar problemas serios.
- Ciertos tipos específicos de error tienen capacidades dedicadas a manipularlos. Por ejemplo, cuando **`new`** falla al asignar memoria, esto puede provocar la ejecución de la función **`new_handler`** para lidiar con el error. Esta función se puede variar al proporcionar un nombre de función como el argumento de **`set_new_handler`**. En la sección 23.14, explicaremos con detalle la función **`set_new_handler`**.

## 23.4 Fundamentos del manejo de excepciones en C++: `try`, `throw` y `catch`

La manipulación de excepciones de C++ está diseñada para situaciones en las que la función que detecta un error es incapaz de lidiar con él. Dicha función arrojará una *excepción*. No existe garantía de que exista "algo allá afuera", es decir, un *manipulador de excepciones*, específicamente diseñado para procesar ese tipo de excepción. Si existe, la excepción será *atrapada* y *manipulada*. Si no existe un manipulador de excepciones para ese tipo en particular de excepción, el programa terminará.

El programador encierra dentro de un *bloque* ***`try`*** el código que podría generar un error que produciría una excepción. El bloque **`try`** va seguido por uno o más bloques **`catch`**. Cada bloque **`catch`** contiene un manipulador de excepciones. Si la excepción coincide con el tipo de parámetro en uno de los bloques **`catch`**, se ejecuta el código para ese bloque **`catch`**. Si no se encuentra un manipulador, se llama a la función **`terminate`**, la cual llama de manera predeterminada a la función **`abort`**.

El control del programa en una excepción lanzada abandona el bloque **`try`** y busca el manipulador apropiado dentro de los bloques **`catch`**. (Pronto explicaremos qué es lo que hace "apropiado" a un manipulador.) Si no se lanzan excepciones dentro de un bloque **`try`**, se ignoran los manipuladores de excepciones para ese bloque y el programa continúa la ejecución después del último bloque **`catch`**.

Podemos especificar las excepciones que lanza una función. Como una opción, podemos especificar si una función debe o no lanzar alguna excepción.

La excepción se lanza dentro de un bloque **`try`** en la función, o la excepción se lanza desde una función llamada directa o indirectamente desde el bloque **`try`**. Al punto en el que ***`throw`*** se ejecuta se le llama *punto de lanzamiento*. Este término también se utiliza para describir a la propia expresión **`throw`**. Una vez que se lanza una excepción, el control no puede regresar al punto de lanzamiento.

Cuando ocurre una excepción, es posible comunicar información al manipulador de excepciones desde el punto de la excepción. Esta información es del tipo del objeto lanzado o información colocada en el objeto lanzado.

Por lo general, el objeto lanzado es una cadena de caracteres (para un mensaje de error) o un objeto de una clase. El objeto lanzado transmite la información al manipulador de excepciones que procesará dicha excepción.



**Observación de ingeniería de software 23.4**

*Una clave para la manipulación de excepciones es que la porción de un programa o sistema que manipulará la excepción puede ser bastante diferente o distante de la porción del programa que detectó y generó la situación excepcional.*

## 23.5 Un ejemplo sencillo de manejo de excepciones: La división entre cero

Ahora consideremos un ejemplo sencillo de manipulación de excepciones. En la figura 23.1 utilizamos **`try`**, **`throw`** y **`catch`** para detectar una división entre cero, para indicar una excepción de división entre cero y para manipular una excepción de división entre cero.

```
// Figura 23.1: fig23_01.cpp
// Un ejemplo sencillo de manejo de excepciones.
// Verificación de una excepción de división entre cero.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// Clase ExcepcionDeDivisionEntreCero a utilizarse en el manejo de
// excepciones para lanzar una excepción sobre una división entre cero.
class ExcepcionDeDivisionEntreCero {
public:
   ExcepcionDeDivisionEntreCero()
      : mensaje( “se intento una division entre cero” ) { }
   const char *what() const { return mensaje; }
private:
   const char *mensaje;
}; // fin de la clase ExcepcionDeDivisionEntreCero

// Definición de la función cociente. Muestra el lanzamiento
// de una excepción cuando se encuentra una división entre cero.
double cociente( int numerador, int denominador )
{
   if ( denominador == 0 )
      throw ExcepcionDeDivisionEntreCero();

   return static_cast< double > ( numerador ) / denominador;
} // fin de la función cociente

// Programa controlador
int main()
{
   int numero1, numero2;
   double resultado;

   cout << “Introduzca dos enteros (fin de archivo para terminar): “;

   while ( cin >> numero1 >> numero2 ) {
```

**Figura 23.1** Un ejemplo sencillo de manipulación de excepciones sobre la división entre cero. (Parte 1 de 2.)

```
40
41        // el bloque try block envuelve el código que podría lanzar una
42        // excepción y el código que no debe ejecutarse
43        // si ocurre una excepción
44        try {
45           resultado = cociente( numero1, numero2 );
46           cout << "El cociente es: " << resultado << endl;
47        } // fin de try
48        catch ( ExcepcionDeDivisionEntreCero ex ) { // manipulador de
                                                       excepciones
49           cout << "Ocurrio una excepcion: " << ex.what() << '\n';
50        } // fin de catch
51
52        cout << "\nIntroduzca dos enteros (fin de archivo para terminar): ";
53     } // fin de while
54
55     cout << endl;
56     return 0;      // termina de manera normal
57  } // fin de la función main
```

```
Introduzca dos enteros (fin de archivo para terminar): 100 7
El cociente es: 14.2857

Introduzca dos enteros (fin de archivo para terminar): 100 0
Ocurrio una excepcion: se intento un division entre cero

Introduzca dos enteros (fin de archivo para terminar): 33 9
El cociente es: 3.66667

Introduzca dos enteros (fin de archivo para terminar): ^Z
```

**Figura 23.1** Un ejemplo sencillo de manipulación de excepciones sobre la división entre cero. (Parte 2 de 2.)

Consideremos ahora el programa controlador en **main**. Observe la declaración "localizada" de **numero1**, y **numero2**.

El programa contiene un bloque **try** (línea 44), el cual contiene el código que podría lanzar la excepción. Observe que la división real que puede provocar el error no se lista explícitamente dentro del bloque **try**. En lugar de ello, la llamada a la función **cociente** contiene el código que en realidad intenta la división. La función **cociente** (definida en la línea 23) en realidad lanza el objeto de excepción de la división entre cero, como veremos más adelante. Por lo general, los errores pueden salir a la superficie a través del código específico mencionado en el bloque **try**, a través de llamadas a una función o incluso a través de llamadas a funciones profundamente anidadas iniciadas por código dentro de un bloque **try**.

El bloque **try** va seguido inmediatamente por el bloque **catch** que contiene un manipulador de excepción para el error de división entre cero. Por lo general, cuando se lanza una excepción dentro de un bloque **try**, la excepción se captura en el bloque **catch,** que especifica el tipo apropiado que coincide con la excepción lanzada. En la figura 23.1, el bloque **catch** especifica que atrapará objetos de tipo **ExcepcionDeDivisionEntreCero**; este tipo coincide con el tipo del objeto lanzado en la función **cociente**. El cuerpo de este manipulador de excepción imprime el mensaje de error devuelto por la llamada a la función **what**. Los manipuladores de excepción pueden ser mucho más elaborados que éste.

Si cuando se ejecuta el código dentro de un bloque **try**, éste no lanza una excepción, entonces todos los manipuladores **catch** inmediatamente después del bloque **try** se ignoran y la ejecución continúa con la siguiente línea de código después de los manipuladores **catch**; en la figura 23.1, si la ejecución de una instrucción **return** devuelve **0**, indica una terminación normal.

Ahora, examinemos las definiciones de la clase **`ExcepcionDeDivisionEntreCero`** y de la función **`cociente`**. En la función **`cociente`**, cuando la instrucción **`if`** determina que el denominador es cero, el cuerpo de dicha instrucción lanza una instrucción **`throw`** que especifica el nombre del constructor para el objeto de la excepción. Esto provoca la creación de un objeto de la clase **`ExcepcionDeDivisionEntreCero`**. Este objeto será atrapado por la instrucción **`catch`** (que especifica el tipo **`ExcepcionDeDivisionEntreCero`**) después del bloque **`try`**. El constructor de la clase **`ExcepcionDeDivisionEntreCero`** simplemente dirige el dato miembro **`mensaje`** hacia la cadena **`“se intento una division entre cero”`**. El objeto lanzado es recibido en el parámetro especificado en el manipulador **`catch`** (en este caso, el parámetro **`ex`**), y el mensaje se imprime ahí a través de una llamada a la función **`what`**.



**Buena práctica de programación 23.4**

*Asociar cada tipo de error en tiempo de ejecución con el nombre del objeto de excepción apropiado, mejora la claridad del programa.*

## 23.6 Cómo arrojar una excepción

La palabra reservada **`throw`** se utiliza para indicar la ocurrencia de una excepción. A esto se le conoce como *lanzar una excepción*. Por lo general, **`throw`** especifica un operando. (Un caso especial que no especifica operandos lo explicaremos más adelante.) El operando de **`throw`** puede ser de cualquier tipo. Si el operando es un objeto, lo llamamos un *objeto de excepción*. El valor de cualquier expresión puede lanzarse en lugar de un objeto. Es posible lanzar objetos que no estén formulados para la manipulación de excepciones.

¿En dónde se atrapa una excepción? Al ser lanzada, la excepción será atrapada por el manipulador de excepciones más cercano (correspondiente al bloque **`try`** a partir del cual se lanzó la excepción), especificando un tipo apropiado. Los manipuladores de excepciones para un bloque **`try`** se listan inmediatamente después del bloque **`try`**.

Como parte del lanzamiento de una excepción, se crea y se inicializa una copia del operando de **`throw`**. Después, este objeto inicializa el parámetro del manipulador de excepciones. El objeto temporal se destruye cuando el manipulador de excepciones completa su ejecución y sale.



**Observación de ingeniería de software 23.5**

*Si es necesario pasar información acerca del error que provocó la excepción, dicha información puede colocarse en el objeto lanzado. El manipulador **`catch`** contendrá entonces un nombre de parámetro a través del cual se puede hacer referencia a esa información.*



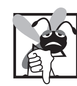
**Observación de ingeniería de software 23.6**

*Un objeto puede lanzarse sin que contenga información a pasar; en este caso, el sólo saber que se lanzó una excepción de este tipo puede proporcionar suficiente información para que el manipulador haga su trabajo correctamente.*

Cuando se lanza una excepción, el control sale del bloque **`try`** actual y continúa con un manipulador **`catch`** apropiado (si existe alguno) después del bloque **`try`**. Es posible que el punto de lanzamiento se encuentre dentro de un alcance profundamente anidado dentro del bloque **`try`**; aun así, el control continuará con el manipulador **`throw`**. También es posible que el punto de lanzamiento pudiera estar dentro de una llamada a función profundamente anidada; aun así, el control continuará con al manipulador **`catch`**.

Es posible que aparezca un bloque **`try`** que no contenga verificación alguna de error, y que no incluya instrucciones **`throw`**, pero el código referenciado en el bloque **`try`** ciertamente podría provocar la ejecución de código de verificación de errores en el constructor. El código en un bloque **`try`** podría realizar una colocación de subíndices a un arreglo en un objeto de clase arreglo, cuya función miembro **`operador[]`** se sobrecarga para lanzar una excepción para un error de subíndice fuera de rango. Cualquier llamada a una función puede invocar código que pudiera lanzar una excepción o una llamada a otra función que lance una excepción.

Aunque una excepción puede terminar la ejecución del programa, no es necesario que lo haga. Sin embargo, una excepción no termina en el bloque en el que ocurrió la excepción.



**Error común de programación 23.3**

*Las excepciones sólo deben lanzarse dentro de un bloque **`try`**. Una excepción lanzada fuera de un bloque **`try`** provoca una llamada a la función **`terminate`**.*



**Error común de programación 23.4**

*Es posible lanzar una excepción condicional. Pero tenga cuidado, ya que las reglas de promoción pueden provocar que el valor devuelto por la expresión condicional sea de un tipo diferente al que usted espera. Por ejemplo, cuando se lanza un `int` o un `double` desde la misma expresión condicional, la expresión condicional convertirá el `int` en `double`. Por lo tanto, el resultado siempre será atrapado mediante `catch` con un argumento `double`, en lugar de atraparlo solamente algunas veces como `double` (para el `double` real), y algunas veces atraparlo como `int`.*

## 23.7 Cómo atrapar una excepción

Los manipuladores de excepciones están contenidos en bloques `catch`. Cada bloque `catch` comienza con la palabra reservada `catch` seguida por paréntesis que contienen un tipo (que indica el tipo de excepción que manipula este bloque `catch`), y un nombre de parámetro opcional. A esto le siguen las llaves que delinean el código de manipulación de la excepción. Cuando se atrapa una excepción, se ejecuta el código en el bloque `catch`.

El manipulador `catch` define su propio alcance. Un `catch` especifica entre paréntesis el tipo del objeto a atrapar. El parámetro en el manipulador `catch` puede o no tener nombre. Si el parámetro tiene nombre, puede hacerse referencia a él en el manipulador. Si el parámetro no tiene nombre (es decir, solamente se lista un tipo para propósitos de coincidencia con el tipo del objeto lanzado), entonces la información no se transmite desde el punto de lanzamiento hacia el manipulador; solamente se pasa el control desde el punto de lanzamiento hacia el manipulador. Para muchas excepciones esto es aceptable.



**Error común de programación 23.5**

*Especificar una lista separada por comas para los argumentos de `catch`, es un error de sitaxis.*

Una excepción cuyos tipos de objetos lanzados coincide con el tipo de los argumentos del encabezado de `catch` provoca la ejecución del bloque `catch`, es decir, que el manipulador para las excepciones de ese tipo se ejecute.

El manipulador `catch` que atrapa una excepción es el primero en la lista después del bloque activo `try` actual que coincide con el tipo del objeto lanzado. Más adelante explicaremos las reglas de coincidencia.

Una excepción que no se atrapa provoca una llamada a `terminate`, la cual termina un programa de manera predeterminada mediante la llamada a `abort`. Es posible especificar un comportamiento personalizado, diseñando otra función a ejecutar si se proporciona el nombre de esa función como el argumento dentro de una llamada a la función `set_terminate`.

Un `catch` seguido por paréntesis con puntos suspensivos

```
catch( ... )
```

significa atrapar todas las excepciones.



**Error común de programación 23.6**

*Colocar `catch( ... )` antes de otros bloques `catch` evita la ejecución de dichos bloques; `catch( ... )` debe colocarse al final de la lista de los manipuladores que siguen al bloque `try`.*



**Observación de ingeniería de software 23.7**

*Una debilidad que se presenta al atrapar excepciones por medio de `catch( ... )` es que, por lo general, no se puede asegurar de qué tipo de excepción se trata. Otra debilidad es que sin un parámetro con nombre, no existe forma de hacer referencia al objeto de excepción dentro del manipulador de excepciones.*

Es posible que ningún manipulador coincida con un objeto en particular lanzado. Esto provoca la búsqueda de una coincidencia para continuar en el siguiente bloque `try` contenido. Al continuar este proceso, en algún momento se determinará que no existe un manipulador dentro del programa que coincida con el tipo del objeto lanzado; en este caso, se llama a la función `terminate`, la cual llama a la función `abort` de manera predeterminada.

Los manipuladores de excepciones se buscan en orden para una coincidencia apropiada. El primer manipulador que arroje una coincidencia se ejecuta. Cuando el manipulador termina su ejecución, el control continúa

con la primera instrucción después del último bloque **`catch`** (es decir, la primera instrucción después del último manipulador de excepción para ese bloque **`try`**).

Es posible que muchos manipuladores de excepciones proporcionen una coincidencia aceptable para el tipo de excepción que se arrojó. En este caso, se ejecuta el primer manipulador de excepción que coincide con el tipo de la excepción. Si coinciden varios manipuladores, y si cada uno de éstos manipula de modo diferente a la excepción, entonces el orden de los manipuladores afectará la manera en que se manipula una excepción.

Es posible que varios manipuladores **`catch`** puedan contener un tipo de clase que coincida con el tipo particular de objeto lanzado. Esto puede suceder por distintas razones. Primero, puede existir un manipulador **`catch ( ... )`** "atrapa todo" que atrapará cualquier excepción. Segundo, debido a las jerarquías de herencia, es posible que un objeto de una clase derivada pueda ser atrapado por el manipulador que especifica el tipo de la clase derivada, o por los manipuladores que especifican los tipos de cualesquiera de las clases base de dicha clase derivada.



#### Error común de programación 23.7

*Colocar un **`catch`** que atrapa un objeto de una clase base antes de un **`catch`** que atrapa un objeto de la clase derivada a partir de la clase base, es un error de lógica. El **`catch`** de la clase base atrapará a todos los objetos de la clase derivada de dicha clase base, por lo que nunca se ejecutará el **`catch`** de la clase derivada.*

#### Tip para prevenir errores 23.1

*El programador determina el orden en el cual se listan los manipuladores de excepciones. Este orden puede afectar la forma en que se manipulan las excepciones originadas en ese bloque **`try`**. Si usted obtiene un comportamiento inesperado en la manipulación de las excepciones de su programa, podría deberse a que el bloque **`catch`** anterior está interceptando y manipulando las excepciones antes de que alcancen el manipulador que les corresponde.*

Algunas veces, los programas pueden procesar muchos tipos de excepciones íntimamente relacionadas. En lugar de proporcionar clases de excepciones separadas y manipuladores **`catch`** para cada una, el programador puede proporcionar una sola clase de excepción y un solo manipulador **`catch`** para un grupo de excepciones. Al ocurrir cada excepción, puede crearse el objeto de excepción con diferentes datos privados. El manipulador **`catch`** puede examinar estos datos privados para distinguir el tipo de las excepciones.

¿Cuándo ocurre una coincidencia? El tipo de parámetro del manipulador **`catch`** coincide con el tipo del objeto lanzado si:

- Son realmente del mismo tipo.
- El tipo de parámetro del manipulador **`catch`** es una clase base pública de la clase del objeto lanzado.
- El parámetro del manipulador es un apuntador de la clase base o un tipo de referencia y el objeto arrojado es un apuntador de una clase derivada o un tipo de referencia.
- El manipulador **`catch`** es de la forma **`catch(...)`**.



#### Error común de programación 23.8

*Colocar un manipulador de excepciones con un argumento de tipo **`void *`** antes de los manipuladores de excepción con otros tipos de apuntadores, provoca un error de lógica. El manipulador **`void`** podría atrapar todas las excepciones de los tipos de apuntadores, de modo que los manipuladores nunca se ejecutarían. Solamente **`catch( ... )`** puede seguir a **`catch( void *)`**.*

Una coincidencia exacta de tipos es necesaria. No se realiza promoción o conversión alguna, cuando se busca una excepción para conversiones de clases derivadas a clases base.

Es posible lanzar objetos **`const`**. En este caso, el tipo del argumento del manipulador **`catch`** también debe declararse como **`const`**.

Si no encuentra un manipulador para una excepción, el programa termina. Aunque esto parezca aceptable, no es lo que los programadores están acostumbrados a hacer. En vez de lo anterior, con frecuencia los errores simplemente suceden y la ejecución del programa continúa, posiblemente sólo "cojeando" un poco.

Un bloque **`try`** seguido por varios **`catchs`** se asemeja a una instrucción **`switch`**. No es necesario utilizar **`break`** para salir de un manipulador de excepción y evitar los manipuladores de excepciones restantes. Cada

bloque **catch** define un alcance distinto, mientras que todos los casos de una instrucción **switch** se encuentran en el alcance de la misma instrucción.



**Error común de programación 23.9**

*Colocar un punto y coma después de un bloque* **try** *o después de un manipulador* **catch** *(además del último* **catch***) seguido de un bloque* **try***, es un error de sintaxis.*

Un manipulador de excepciones no puede acceder automáticamente a los objetos definidos dentro del bloque **try**, ya que, cuando ocurre una excepción, el bloque **try** termina y todos los objetos automáticos dentro del bloque **try** se destruyen antes de comenzar la ejecución del manipulador.

¿Qué sucede cuando ocurre una excepción dentro de un manipulador de excepciones? La excepción original que se atrapó, se manipula de manera oficial cuando comienza la ejecución del manipulador de excepciones. De modo que las excepciones que ocurren dentro de un manipulador de excepciones necesitan procesarse fuera del bloque **try** en el cual se lanzó la excepción original.

Los manipuladores de excepciones pueden escribirse de distintas formas. Podrían echar un vistazo más cercano a un error y decidir llamar a **terminate**. Podrían *relanzar* una excepción (sección 23.8). Podrían realizar cualquier recuperación necesaria y continuar la ejecución después del último manipulador de excepciones. Podrían revisar la situación que provocó el error, eliminar la causa del error y reintentar mediante la llamada a la función original que provocó la excepción. (Esto no crearía una recursividad infinita.) Podrían devolver algún valor de estado a su entorno, etcétera.



**Observación de ingeniería de software 23.8**

*Es mejor incorporar su estrategia de manipulación de excepciones dentro de un sistema, a partir del comienzo del proceso de diseño. Es difícil agregar una manipulación de excepciones efectiva después de que un sistema ya se implementó.*

Cuando un bloque **try** no lanza excepciones, y dicho bloque **try** completa su ejecución normal, el control se pasa a la primera instrucción después del último **catch** a continuación de **try**.

No es posible volver al punto de lanzamiento mediante una instrucción **return** en un manipulador de excepciones. Dicho **return** simplemente regresa a la función que llamó a la función que contiene el bloque **catch**.



**Error común de programación 23.10**

*Asumir que después de procesar una excepción, el control regresará a la primera instrucción después de* **throw***, es un error de lógica.*



**Observación de ingeniería de software 23.9**

*Otra razón para no utilizar las excepciones para el flujo de control normal es que estas “excepciones” adicionales pueden interponerse en el camino de las excepciones genuinas de tipos de error. Para el programador se vuelve más difícil dar seguimiento al número de casos de excepciones. Por ejemplo, cuando un programa procesa una variedad excesiva de excepciones, ¿podemos estar seguros de lo que atrapa un* **catch( ... )***? Las situaciones excepcionales deben ser raras, no comunes.*

Cuando atrapamos una excepción, es posible que los recursos estén almacenados, pero no liberados en el bloque **try**. Si es posible, el manipulador **catch** debe liberar estos recursos. Por ejemplo, un manipulador **catch** debe eliminar espacio almacenado por **new** y debe cerrar cualquier archivo abierto en el bloque **try** que lanzó la excepción.

Un bloque **catch** puede procesar el error de una manera que permite al programa continuar correctamente la ejecución. O el bloque **catch** puede terminar el programa.

Un manipulador **catch** por sí mismo puede descubrir un error y lanzar una excepción. Dicha excepción no se procesará por medio de los manipuladores **catch** asociados con el mismo bloque **try**, mientras el manipulador **catch** lanza la excepción.



**Error común de programación 23.11**

*Asumir que una excepción lanzada desde un manipulador* **catch** *se procesará por medio de dicho manipulador o cualquier otro manipulador asociado con el bloque* **try** *que lanzó la excepción que provocó la ejecución del manipulador* **catch** *original, es un error lógico.*

## 23.8 Cómo relanzar una excepción

Es posible que un manipulador que atrapa una excepción decida que no puede procesar dicha excepción, o simplemente desea liberar los recursos antes de dejar a alguien más manipularlo. En este caso, el manipulador simplemente relanza la excepción con la instrucción

```
throw;
```

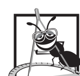
Dicho **throw** sin argumentos relanza la excepción. Si no se lanzó excepción alguna para comenzar, entonces el relanzamiento provoca una llamada a **terminate**.

### Error común de programación 23.12

*Colocar una instrucción* ***throw*** *vacía fuera del manipulador* ***catch****, y ejecutar dicho manipulador, provoca una llamada a* ***terminate****.*

Incluso si un manipulador puede procesar una excepción, sin importar si se lleva a cabo algún proceso en dicha excepción, el manipulador puede relanzar la excepción para futuros procesos fuera del manipulador.

Una excepción relanzada se detecta por medio del siguiente bloque **try**, y se manipula mediante un manipulador de excepciones listado en el bloque **try** que lo contiene.



### Observación de ingeniería de software 23.10

*Utilice* ***catch( ... )*** *para realizar la recuperación que no depende del tipo de excepción, tal como la liberación de recursos comunes. La excepción puede relanzarse para alertar a bloques* ***catch*** *más específicos.*

El programa de la figura 23.2 muestra el relanzamiento de una excepción. En el bloque **try** de **main**, la función **lanzaExcepcion** es llamada en la línea 31. En el bloque **try** de la función **lanzaExcepcion**, la instrucción **throw** de la línea 17 lanza una instancia de la clase **exception** de la biblioteca estándar (definida en el archivo de encabezado **<exception>**. Esta excepción se captura de inmediato en el manipulador **catch** de la línea 19, la cual imprime un mensaje de error, luego relanza la excepción. Esto termina la función **lanzaExcepcion** y devuelve el control al bloque **try/catch** en **main**. La excepción se atrapa de nuevo en la línea 34, y se imprime un mensaje de error.

```
1  // Figura 23.2: fig23_02.cpp
2  // Demostración de un relanzamiento de una excepción.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <exception>
9
10 using std::exception;
11
12 void lanzaExcepcion()
13 {
14    // Lanza una excepción e inmediatamente la atrapa.
15    try {
16       cout << "Funcion lanzaExcepcion\n";
17       throw exception();  // genera una excepción
18    } // fin de try
19    catch( exception e )
20    {
21       cout << "Excepcion manipulada en la funcion lanzaExcepcion\n";
22       throw;  // relanza la excepción para un posterior procesamiento
23    } // fin de catch
24
```

**Figura 23.2** Relanzamiento de una excepción. (Parte 1 de 2.)

```
   cout << "Esto tampoco debe imprimirse\n";
} // fin de la función lanzaExcepcion

int main()
{
   try {
      lanzaExcepcion();
      cout << "Esto no debe imprimirse\n";
   } // fin de try
   catch ( exception e )
   {
      cout << "Excepcion manipulada en main\n";
   } // fin de catch

   cout << "El control del programa continua despues del catch en main"
        << endl;
   return 0;
} // fin de la función main
```

```
Funcion lanzaExcepcion
Excepcion manipulada en la funcion lanzaExcepcion
Excepcion manipulada en main
El control del programa continua despues del catch en main
```

**Figura 23.2** Relanzamiento de una excepción. (Parte 2 de 2.)

## 23.9 Especificaciones de las excepciones

Una *especificación de excepción* enumera una lista de excepciones que puede relanzarse mediante una función que se especifica como:

```
int g( double h ) throw (a, b, c )
{
   // cuerpo de la función
}
```

Es posible restringir los tipos de excepción lanzados desde una función. Los tipos de excepciones se especifican en la declaración de la función como una *especificación de excepción* (también llamada lista de lanzamiento). Las listas de especificación de excepciones listan las excepciones que pueden lanzarse. Una función puede lanzar excepciones indicadas o tipos derivados. No obstante que esto presupone la garantía de que no se lanzarán otras excepciones, es posible hacerlo. Si se lanza una excepción no listada en la especificación de excepciones, se llama a la función **`unexpected`**.

Colocar **`throw()`** (es decir, una *especificación de excepciones* vacía) después de la lista de parámetros de una función, establece que la función no arrojará excepciones. Tal función podría, de hecho, lanzar una excepción; esto también podría generar una llamada a **`unexpected`**.



**Error común de programación 23.13**

*Lanzar una excepción no en la especificación de excepciones de la función, provoca una llamada a* ***`unexpected`***.

Una función sin especificación de excepciones puede lanzar cualquier excepción:

```
void g();   // esta función puede lanzar cualquier excepción
```

El significado de la función **`unexpected`** se puede redefinir al llamar a la función **`set_unexpected`**.

Un aspecto interesante de la manipulación de excepciones es que el compilador no la considerará un error de sintaxis, si una función contiene una expresión **`throw`** para una excepción no listada en la especificación

de excepciones de la función. La función debe intentar lanzar la excepción en tiempo de ejecución antes de que el error sea atrapado.

Si una función lanza una excepción de un tipo de clase en particular, dicha función también puede lanzar excepciones de todas las clases derivadas a partir de dicha clase base, mediante herencia pública.

## 23.10 Cómo procesar excepciones inesperadas

La función **unexpected** llama a la función especificada dentro de la función **set_unexpected**. Si no se especifica función alguna de esta manera, se llama a **terminate** de manera predeterminada.

Se puede llamar a la función **terminate** explícitamente si no se puede atrapar una excepción lanzada, si la pila se corrompe durante la manipulación de excepciones, como la acción predeterminada en una llamada a **unexpected**, y si mientras se desenrolla la pila iniciada por una excepción se intenta lanzar una excepción por medio de un destructor, se provoca la llamada a **terminate**.

La función **set_terminate** puede especificar la función a la que se llamará cuando se llama a la función **terminate**. De lo contrario **terminate** llamará a **abort**.

Los prototipos para las funciones **set_terminate** y **set_unexpected** se localizan en el archivo de encabezado **<exception>**.

La función **set_terminate** y la función **set_unexpected** devuelven cada una un apuntador a la última función llamada por **terminate** y **unexpected**. Esto permite al programador guardar el apuntador a la función, de modo que lo pueda restaurar posteriormente.

Las funciones **set_terminate** y **set_unexpected** toman como argumentos apuntadores a las funciones. Cada argumento debe apuntar a una función con el tipo de retorno **void** sin argumentos.

Si la última acción de una función de terminación definida por el usuario no es la salida del programa, por lo general se llamará automáticamente a la función **abort** para terminar la ejecución del programa, después de la ejecución de las otras instrucciones de la función de terminación definida por el usuario.

## 23.11 Cómo desenrollar una pila

Cuando se lanza una excepción, pero no se atrapa en algún alcance en particular, la llamada a la función pila se desenrolla y se intenta atrapar a la excepción en el siguiente bloque **try/catch** más externo. Desenrollar la llamada a la función pila significa que termina la función en la cual no se atrapó a la excepción, que todas las variables locales en dicha función se destruyen y que el control regresa al punto en el que se llamó a la función. Si ese punto en el programa es un bloque **try**, se intenta atrapar a la excepción. Si ese punto en el programa no es un bloque **try** o no se atrapa la excepción, de nuevo se desenrolla la pila. Como explicamos en la sección anterior, si no se atrapa la excepción en el programa, éste llama a la función **terminate**. El programa de la figura 23.3 muestra cómo desenrollar una pila.

```
// Figura 23.3: fig23_03.cpp
// Demostración de cómo desenrollar una pila.
#include <iostream>

using std::cout;
using std::endl;

#include <stdexcept>

using std::runtime_error;

void funcion3() throw ( runtime_error )
{
   throw runtime_error( “runtime_error en funcion3” );
```

**Figura 23.3** Demostración de cómo desenrollar una pila. (Parte 1 de 2.)

```
15 } // fin de la función function3
16
17 void funcion2() throw ( runtime_error )
18 {
19    funcion3();
20 } // fin de la función function2
21
22 void funcion1() throw ( runtime_error )
23 {
24    funcion2();
25 } // fin de la función function1
26
27 int main()
28 {
29    try {
30       funcion1();
31    } // fin de try
32    catch ( runtime_error e )
33    {
34       cout << "Ocurrio una excepcion: " << e.what() << endl;
35    } // fin de catch
36
37    return 0;
38 } // fin de la función main
```

```
Ocurrio una excepcion: runtime_error en funcion3
```

**Figura 23.3** Demostración de cómo desenrollar una pila. (Parte 2 de 2.)

En **main**, el bloque **try** llama a **funcion1** (línea 30). A continuación, **funcion1** (definida en la línea 22) llama a **función2**. Después, **funcion2** (definida en la línea 17) llama a **funcion3**. La línea 14 de **funcion3** lanza un objeto de excepción. La línea 14 no es un bloque **try**, de modo que se desenrolla la pila, **funcion3** termina en la línea 19 y el control regresa a **funcion2**. La línea 19 no es un bloque **try**, de modo que de nuevo se desenrolla la pila, **función2** termina en la línea 24 y el control regresa a **funcion1**. La línea 24 no es un bloque **try**, de modo que una vez más se desenrolla la pila, **funcion1** termina en la línea 30 y el control regresa a **main**. La línea 30 es un bloque **try**, de modo que la excepción puede atraparse y procesarse en el primer manipulador **catch** coincidente después del bloque **try** (en la línea 32).

## 23.12 Constructores, destructores y manejo de excepciones

Primero, consideremos un tema que habíamos mencionado, pero que aún no hemos resuelto satisfactoriamente: ¿qué sucede cuando se detecta un error dentro de un constructor? Por ejemplo, ¿cómo debe responder un constructor de **Cadena** cuando **new** falla e indica que fue incapaz de obtener el espacio necesario para almacenar la representación interna de **Cadena**? El problema es que un constructor no devuelve valor alguno, entonces ¿cómo le hacemos saber al mundo exterior que el objeto no se construyó apropiadamente? Un método es simplemente devolver el objeto construido de manera inapropiada y esperar que alguien que utilice el objeto haga las pruebas apropiadas para determinar que el objeto estaba mal. Otro método es establecer alguna variable fuera del constructor. Una excepción lanzada pasa la información acerca del constructor que falló hacia el mundo exterior y la responsabilidad de lidiar con la falla.

Para atrapar una excepción, el manipulador de excepciones debe tener acceso a un constructor de copia para el objeto lanzado. (También es válida la copia predeterminada de miembros.)

Las excepciones lanzadas en los constructores provocan que se llame a los destructores para cualquier objeto construido como parte del objeto que se construyó antes del lanzamiento de la excepción.

Los destructores se llaman en cada objeto automático construido dentro de un bloque **try**, antes de que se lance una excepción. Una excepción se manipula en el momento en el que comienza la ejecución del manipulador; en ese punto se garantiza que la pila se desenrolle completamente. Si un destructor invocado como resultado de la pila desenrollada arroja una excepción, se llama a **terminate**.

Si un objeto tiene objetos miembro y si la excepción se lanza antes de la construcción completa del objeto externo, se ejecutarán los destructores de los objetos miembro que se construyeron completamente antes de que ocurriera la excepción.

Si un arreglo de objetos se construyó parcialmente al ocurrir una excepción, solamente se llamará a los destructores de los elementos construidos del arreglo.

Una excepción podría impedir una operación de código que libera un recurso, y provocar así una *fuga de recursos*. Una técnica para resolver este problema es inicializar un objeto local cuando se adquiere el recurso. Cuando ocurre una excepción, se invocará al destructor y se podrá liberar dicho recurso.

Es posible atrapar excepciones lanzadas desde destructores, encerrando a la función que llama al destructor dentro de un bloque **try** y proporcionando un manipulador **catch** con el tipo apropiado. Después de que el manipulador de excepciones completa su ejecución, se ejecuta el destructor del objeto lanzado.

## 23.13 Excepciones y herencia

Distintas clases de excepciones pueden derivarse a partir de una clase base común. Si un **catch** atrapa un apuntador o una referencia a un objeto de excepción de un tipo de clase base, también puede atrapar un apuntador o una referencia a todos los objetos de las clases derivadas a partir de la clase base. Esto puede permitir el procesamiento polimórfico de errores relacionados.

**Tip para prevenir errores 23.2**

*Utilizar la herencia con excepciones permite a un manipulador de excepciones atrapar errores relacionados por medio de dos notaciones más concisas. Ciertamente, una podría atrapar individualmente a cada tipo de apuntador o referencia a una excepción de clase derivada, pero es más conciso atrapar apuntadores o referencias a los objetos de excepción de una clase base. Además, atrapar individualmente apuntadores o referencias a objetos de excepciones de clase derivadas es causa de errores, si el programador olvida probar explícitamente uno o más de los tipos de apuntadores o referencias a clases derivadas.*

## 23.14 Cómo procesar fallas de new

Existen varios métodos para lidiar con las fallas de **new**. Hasta este punto, utilizamos la macro **assert** para probar el valor devuelto por **new**. Si dicho valor es **0**, la macro **assert** termina el programa. Éste no es un mecanismo robusto para lidiar con las fallas de **new** (no nos permite recuperar la falla de manera alguna). El C++ estándar especifica que cuando **new** falla, lanza una excepción **bad_alloc** (definida en el archivo de encabezado **<new>**. Sin embargo, es posible que algunos compiladores no cumplan con el estándar de C++ y, por lo tanto, utilicen la versión de **new** que, ante una falla, devuelve **0**. En esta sección presentamos tres ejemplos de la falla de **new**. El primer ejemplo devuelve **0** cuando **new** falla. El segundo y el tercer ejemplo utilizan la versión de **new** que arroja la excepción **bad_alloc** cuando **new** falla.

La figura 23.4 muestra el **new** que devuelve **0** ante una falla, para asignar la cantidad requerida de memoria. Se supone que la estructura **for** de la línea 12 hace un ciclo 50 veces y asigna valores **double** dentro de un arreglo de 5,000,000 de elementos (es decir, 40,000,000 bytes, debido a que por lo general **double** es de 8 bytes) cada vez dentro del ciclo. La estructura **if** de la línea 15 prueba el resultado de cada operación **new** para determinar si la memoria se asignó. Si **new** falla y devuelve **0**, se imprime el mensaje **“Memory allocation failed”** y el ciclo termina.

```
1 // Figura 23.4: fig23_04.cpp
2 // Demostración de new devolviendo 0
3 // cuando la memoria no se asigna
4 #include <iostream>
```

**Figura 23.4** Demostración de un **new** que devuelve cero ante una falla. (Parte 1 de 2.)

```

using std::cout;

int main()
{
   double *ptr[ 50 ];

   for ( int i = 0; i < 50; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      if ( ptr[ i ] == 0 ) { // new falló al asignar la memoria
         cout << "La asignacion de memoria fallo en ptr[ "
              << i << " ]\n";
         break;
      } // fin de if
      else
         cout << "5000000 doubles asignados en ptr[ "
              << i << " ]\n";
   } // fin de for

   return 0;
} // fin de la función main
```

```
5000000 doubles asignados en ptr[ 0 ]
5000000 doubles asignados en ptr[ 1 ]
5000000 doubles asignados en ptr[ 2 ]
5000000 doubles asignados en ptr[ 3 ]
5000000 doubles asignados en ptr[ 4 ]
5000000 doubles asignados en ptr[ 5 ]
5000000 doubles asignados en ptr[ 6 ]
5000000 doubles asignados en ptr[ 7 ]
5000000 doubles asignados en ptr[ 8 ]
5000000 doubles asignados en ptr[ 9 ]
5000000 doubles asignados en ptr[ 10 ]
5000000 doubles asignados en ptr[ 11 ]
5000000 doubles asignados en ptr[ 12 ]
5000000 doubles asignados en ptr[ 13 ]
5000000 doubles asignados en ptr[ 14 ]
5000000 doubles asignados en ptr[ 15 ]
5000000 doubles asignados en ptr[ 16 ]
5000000 doubles asignados en ptr[ 17 ]
5000000 doubles asignados en ptr[ 18 ]
La asignacion de memoria fallo en ptr[ 19 ]
```

**Figura 23.4** Demostración de un **`new`** que devuelve cero ante una falla. (Parte 2 de 2.)

La salida muestra que tuvieron que realizarse 19 iteraciones del ciclo antes de que **`new`** fallara y terminara el ciclo. Su salida podría diferir dependiendo de la memoria física, el espacio en disco disponible para la memoria virtual de su sistema y del compilador utilizado para compilar un programa.

La figura 23.5 muestra el **`new`** que arroja **`bad_alloc`** cuando falla al asignar la memoria requerida. Se supone que la estructura **`for`** de la línea 18 dentro del bloque **`try`** debe repetir el ciclo 50 veces y en cada pasada asignar un arreglo de 5,000,000 valores **`double`** (es decir, 40,000,000 bytes, debido a que **`double`** por lo general es de 8 bytes). Si **`new`** falla y arroja una excepción **`bad_alloc`**, el ciclo termina y el programa continúa en el flujo de control de la manipulación de excepciones de la línea 24, en donde la excepción se atrapa y se procesa. Se imprime el mensaje **`"Ocurrio una excepcion:"`**, seguido por la cadena (que contiene el

```
// Figura 23.5: fig23_05.cpp
// Demostración de new lanzando bad_alloc
// cuando la memoria no se asigna
#include <iostream>

using std::cout;
using std::endl;

#include <new>

using std::bad_alloc;

int main()
{
   double *ptr[ 50 ];

   try {
      for ( int i = 0; i < 50; i++ ) {
         ptr[ i ] = new double[ 5000000 ];
         cout << “5000000 doubles asignados en ptr[ “
              << i << “ ]\n”;
      } // fin de for
   } // fin de try
   catch (  bad_alloc exception ) {
      cout << “Ocurrio una excepcion: “
           << exception.what() << endl;
   } // fin de catch

   return 0;
} // fin de la función main
```

```
5000000 doubles asignados en ptr[ 0 ]
5000000 doubles asignados en ptr[ 1 ]
5000000 doubles asignados en ptr[ 2 ]
5000000 doubles asignados en ptr[ 3 ]
5000000 doubles asignados en ptr[ 4 ]
5000000 doubles asignados en ptr[ 5 ]
5000000 doubles asignados en ptr[ 6 ]
5000000 doubles asignados en ptr[ 7 ]
5000000 doubles asignados en ptr[ 8 ]
5000000 doubles asignados en ptr[ 9 ]
5000000 doubles asignados en ptr[ 10 ]
5000000 doubles asignados en ptr[ 11 ]
5000000 doubles asignados en ptr[ 12 ]
Ocurrió una excepción: bad_alloc
```

**Figura 23.5** Demostración del **`new`** que lanza **`bad_alloc`** ante una falla.

mensaje específico de la excepción **`“Allocation Failure”`**) devuelto por **`exception.what()`**. La salida muestra que se realizan 12 iteraciones del ciclo antes de que **`new`** fallara y lanzara la excepción **`bad_alloc`**. Su salida podría diferir dependiendo en la memoria física, el espacio en disco disponible para la memoria virtual de su sistema y del compilador que utiliza para compilar el programa.

Los compiladores varían en cuanto al soporte para la manipulación de fallas de **`new`**. Muchos compiladores de C++ devuelven **`0`** de manera predeterminada cuando **`new`** falla. Algunos de estos compiladores soportan el **`new`** que arroja la excepción, si se incluye el archivo de encabezado **`<new>`** (o **`<new.h>`**). Otros compila-

dores arrojan **bad_alloc** de manera predeterminada, sin importar si usted incluye el archivo de encabezado **<new>**. Lea la documentación de su compilador para determinar el soporte de su compilador para la manipulación de fallas de **new**.

El estándar de C++ especifica que los compiladores que cumplen con el estándar aún pueden utilizar una versión de **new** que devuelve **0** cuando falla. Para este propósito, el archivo de encabezado **<new>** define **nothrow** (de tipo **nothrow_t**), el cual se utiliza de la siguiente manera:

```
double *ptr = new( nothrow ) double[ 5000000 ]
```

La instrucción anterior indica que la versión de **new** que no relanza excepciones **bad_alloc** (es decir, **nothrow**) debe utilizarse para asignar un arreglo de 5,000,000 **doubles**.



### Observación de ingeniería de software 23.11

*El estándar de C++ recomienda que para hacer programas más robustos, los programadores deben utilizar la versión de **new** que lanza excepciones **bad_alloc** ante una falla.*

Existe una característica adicional que puede utilizar para manipular las fallas de **new**. La función **set_new_handler** (cuyo prototipo se encuentra en el archivo de encabezado **<new>**) toma como su argumento un apuntador a una función para la función que no toma argumentos, y devuelve **void**. El apuntador a la función se registra como la función a llamar cuando **new** falla. Esto proporciona al programador un método uniforme para procesar cada falla de **new** sin importar en dónde ocurre dicha falla en el programa. Una vez que en el programa se registra un *manipulador **new*** con **set_new_handler**, **new** no lanza **bad_alloc** ante una falla.

El operador **new** es en realidad un ciclo que intenta adquirir memoria. Si la memoria se asigna, **new** devuelve un apuntador a dicha memoria. Si **new** falla al asignar la memoria y se registró la función de manipulación de **new**, se llama a la nueva función de manipulación de **new**. El estándar de C++ especifica que la nueva función de **new** debe realizar una de las siguientes tareas:

1. Haga que más memoria esté disponible, eliminando otra memoria asignada dinámicamente y regrese al ciclo en el operador **new** para intentar asignar de nuevo la memoria.
2. Lance una excepción de tipo **bad_alloc**.
3. Llame a la función **abort** o **exit** (ambas del archivo de encabezado **<cstdlib>**) para terminar el programa.

La figura 23.6 muestra **set_new_handler**. La función **personalizaNuevoManip** simplemente imprime un mensaje de error y termina el programa con una llamada a **abort**. La salida muestra sólo 11 iteraciones del ciclo durante la ejecución antes de que falle **new** y lance la excepción **bad_alloc**. Su salida puede diferir, dependiendo de la memoria física, el espacio en disco disponible para la memoria virtual en su sistema y el compilador que usted utiliza para compilar el programa.

```
// Figura 23.6: fig23_06.cpp
// Demostración del manipulador set_new_handler
#include <iostream>

using std::cout;
using std::cerr;

#include <new>
#include <cstdlib>

using std::set_new_handler;

void personalizaNuevoManip()
{
   cerr << "se llamo a personalizaNuevoManip";
   abort();
```

**Figura 23.6** Demostración de **set_new_handler**. (Parte 1 de 2.)

```
} // fin de la función personalizaNuevoManip

int main()
{
   double *ptr[ 50 ];
   set_new_handler( personalizaNuevoManip );

   for ( int i = 0; i < 50; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      cout << “5000000 doubles asignados en ptr[ “
           << i << “ ]\n”;
   } // fin de for

   return 0;
} // fin de la función main
```

```
5000000 doubles asignados en ptr[ 0 ]
5000000 doubles asignados en ptr[ 1 ]
5000000 doubles asignados en ptr[ 2 ]
5000000 doubles asignados en ptr[ 3 ]
5000000 doubles asignados en ptr[ 4 ]
5000000 doubles asignados en ptr[ 5 ]
5000000 doubles asignados en ptr[ 6 ]
5000000 doubles asignados en ptr[ 7 ]
5000000 doubles asignados en ptr[ 8 ]
5000000 doubles asignados en ptr[ 9 ]
5000000 doubles asignados en ptr[ 10 ]
5000000 doubles asignados en ptr[ 11 ]
se llamo a personalizaNuevoManip
```

**Figura 23.6** Demostración de **`set_new_handler`**. (Parte 2 de 2.)

## 23.15 La clase `auto_ptr` y la asignación dinámica de memoria

Una práctica común de programación es asignar memoria dinámicamente (posiblemente un objeto) en un espacio vacío, asignar la dirección de dicha memoria a un apuntador, utilizar el apuntador para manipular la memoria y desalojar la memoria con **`delete`** cuando la memoria ya no es necesaria. Si ocurre una excepción después de asignar la memoria y antes de la ejecución de la instrucción **`delete`**, entonces podría ocurrir una fuga de memoria. El estándar de C++ proporciona la plantilla de clase **`auto_ptr`** en el archivo de encabezado **`<memory>`**, para lidiar con esta situación.

Un objeto de clase **`auto_ptr`** mantiene un apuntador a la memoria asignada dinámicamente. Cuando un objeto **`auto_ptr`** sale de alcance, realiza una operación **`delete`** en su dato miembro apuntador. La plantilla de la clase **`auto_ptr`** proporciona los operadores **`*`** y **`->`** de modo que un objeto **`auto_ptr`** puede utilizarse como una variable de apuntador normal. La figura 23.7 muestra un objeto **`auto_ptr`** que apunta a un objeto de la clase **`Entero`** (definida en las líneas 12 a 22).

```
1  // Figura 23.7: fig23_07.cpp
2  // Demostración de auto_ptr
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
```

**Figura 23.7** Demostración de **`auto_ptr`**. (Parte 1 de 2.)

```
7
8  #include <memory>
9
10 using std::auto_ptr;
11
12 class Entero {
13 public:
14    Entero( int i = 0 ) : valor( i )
15       { cout << “Constructor para Entero “ << valor << endl; }
16    ~Entero()
17       { cout << “Destructor para Entero “ << valor << endl; }
18    void estableceEntero( int i ) { valor = i; }
19    int obtieneEntero() const { return valor; }
20 private:
21    int valor;
22 }; // fin de la clase Entero
23
24 int main()
25 {
26    cout << “Creando un objeto auto_ptr que apunta “
27         << “hacia un Entero\n”;
28
29    auto_ptr< Entero > ptrHaciaEntero( new Entero( 7 ) );
30
31    cout << “Utilizando auto_ptr para manipular el Entero\n”;
32    ptrHaciaEntero->estableceEntero( 99 );
33    cout << “Entero despues de estableceEntero: “
34         << ( *ptrHaciaEntero ).obtieneEntero()
35         << “\nTerminando programa” << endl;
36
37    return 0;
38 } // fin de la función main
```

```
Creando un objeto auto_ptr que apunta hacia un Entero
Constructor para Entero 7
Utilizando auto_ptr para manipular el Entero
Entero despues de estableceEntero: 99
Terminando programa
Destructor para Entero 99
```

**Figura 23.7** Demostración de **`auto_ptr`**. (Parte 2 de 2.)

La línea 29

```
auto_ptr< Entero > ptrHaciaEntero( new Entero( 7 ) );
```

crea un objeto **`auto_ptr`** llamado **`ptrHaciaEntero`** y lo inicializa con un apuntador a un objeto **`Entero`** asignado dinámicamente, el cual contiene el valor 7.

La línea 32

```
ptrHaciaEntero->estableceEntero( 99 );
```

utiliza el operador **`->`** de **`auto_ptr`** y el operador de llamada a función **`()`** para llamar a la función **`estableceEntero`** en el objeto **`Entero`** al que apunta **`ptrHaciaEntero`**. La llamada

```
( *ptrHaciaEntero ).obtieneEntero()
```

de la línea 34 utiliza el operador sobrecargado **`*`** **`auto_ptr`** para desreferenciar a **`ptrHaciaEntero`**, después utiliza el operador punto (**`.`**) y el operador de llamada a función **`()`** para llamar a la función **`obtieneEntero`** en el objeto **`Entero`** al que apunta **`ptrHaciaEntero`**.

La variable **`ptrHaciaEntero`** es una variable local automática en **`main`**, de modo que **`ptrHaciaEntero`** se destruye cuando **`main`** termina. Esto fuerza el **`delete`** del objeto **`Entero`** al que apunta **`ptrHaciaEntero`**, el cual, por supuesto, fuerza una llamada al destructor de la clase **`Entero`**. Lo más importante, esta técnica puede prevenir las fugas de memoria.

## 23.16 Jerarquía de la biblioteca estándar de excepciones

La experiencia muestra que las excepciones caen en cierto número de categorías. El estándar de C++ incluye una jerarquía de clases de excepción. Esta jerarquía es encabezada por la clase base (definida en el archivo de encabezado **`<exception>`**), la cual contiene la función **`what()`** que se acarrea en cada clase derivada para emitir el mensaje de error apropiado.

A partir de la clase **`exception`**, algunas de las clase derivadas inmediatas son **`runtime_error`** y **`logic_error`** (ambas se definen en el encabezado **`<stdexcept>`**), cada una de las cuales tiene varias clases derivadas.

También derivadas de **`exception`** son las excepciones lanzadas por las características del lenguaje de C++, por ejemplo, **`new`** lanza **`bad_alloc`** (sección 23.14), **`dynamic_cast`** lanza **`bad_cast`** y **`typeid`** lanza **`bad_typeid`**. Al incluir **`std::bad_exception`** en la lista de lanzamientos de una función, si ocurre una excepción inesperada, **`unexpected()`** puede arrojar **`bad_exception`**, en lugar de terminar (de manera predeterminada), o en lugar de llamar a otra función especificada con **`set_unexpected`**.

La clase **`logic_error`** es la clase base de varias clases de excepción estándar que indican errores en la lógica del programa que con frecuencia pueden prevenirse escribiendo el código apropiado. Continuamos con las descripciones de algunas de estas clases. La clase **`invalid_argument`** indica que se pasó un argumento inválido a la función. (El código apropiado puede, por supuesto, evitar los argumentos inválidos al alcanzar una función.) La clase **`length_error`** indica que una longitud mayor que el tamaño máximo permitido para el objeto que se manipula se utilizó para ese objeto. La clase **`out_of_range`** indica que un valor tal como un subíndice dentro de un arreglo de cadena está fuera de rango.

La clase **`runtime_error`** es la clase base para varias otras clases estándar de excepciones que indican errores en un programa y que solamente pueden detectarse en tiempo de ejecución. La clase **`overflow_error`** indica que ocurrió un desbordamiento aritmético. La clase **`underflow_error`** indica que ocurrió un error de insuficiencia aritmética.



**Observación de ingeniería de software 23.12**

*El objetivo de la jerarquía estándar de **`exception`** es que sirva como un punto de inicio. Los usuarios pueden lanzar excepciones estándar, lanzar excepciones derivadas de las excepciones estándar o lanzar sus propias excepciones no derivadas de las excepciones estándar.*



**Error común de programación 23.14**

*Las clases de excepción definidas por el usuario no necesitan derivarse de la clase **`exception`**. Por lo tanto, escribir **`catch( exception e )`** no garantiza el atrapar todas las excepciones que pueda encontrar un programa.*



**Tip para prevenir errores 23.3**

*Para atrapar todas las excepciones que pudieran arrojarse en un bloque **`try`**, utilice **`catch( ... )`**.*

## RESUMEN

- Algunos ejemplos comunes del manejo de excepciones son los subíndices fuera de rango de los arreglos, el desbordamiento aritmético de flujo, la división entre cero, los parámetros no válidos de una función y la insuficiencia de memoria para satisfacer una asignación mediante **`new`**.
- El espíritu detrás del manejo de excepciones es permitir atrapar y manipular los errores, en lugar de simplemente dejarlos ocurrir y sufrir las consecuencias. Con el manejo de excepciones, si el programador no proporciona los medios para manipular un error fatal, el programa terminará; por lo general, los errores no fatales permiten al programador continuar la ejecución, pero producen resultados incorrectos.
- El manejo de excepciones está diseñado para lidiar con errores de sincronización (es decir, errores que ocurren como resultado de la ejecución del programa).

- El manejo de excepciones no está diseñado para lidiar con situaciones asíncronas tales como la llegada de mensajes de red, operaciones de E/S en disco, clics del ratón; éstas se manejan mejor a través de otros medios, tales como el procesamiento de interrupciones.
- Por lo general, el manejo de excepciones se utiliza en situaciones en las que se lidiará con el error en una parte diferente del programa (es decir, con un alcance diferente) a la que detectó el error.
- Las excepciones no deben utilizarse como un mecanismo para especificar el flujo de control. Por lo general, el flujo de control con estructuras de control convencionales es más claro y más eficiente que con las excepciones.
- El manejo de excepciones debe utilizarse para procesar excepciones de los componentes de un programa que no son capaces de manejar excepciones de manera directa.
- El manejo de excepciones debe utilizarse para procesar excepciones de componentes de software tales como funciones, bibliotecas, y clases que puedan utilizarse ampliamente y en donde no tiene sentido que manejen sus propias excepciones.
- El manejo de excepciones debe utilizarse en proyectos grandes para manipular los errores de procesamiento de manera uniforme para todo el proyecto.
- El manejo de excepciones en C++ está diseñado para situaciones en las que una función que detecta un error no es capaz de lidiar con él. Dicha función arroja una excepción. Si la excepción coincide con el tipo del parámetro en uno de los bloques **catch**, éste se ejecuta. De lo contrario se llama a la función **terminate**, la cual llama de manera predeterminada a la función **abort**.
- El programador encierra en un bloque **try** el código que pudiera generar un error que producirá una excepción. El bloque **try** va inmediatamente seguido por uno o más bloques **catch**. Cada bloque **catch** especifica el tipo de excepción que puede atrapar y manipular. Cada bloque **catch** contiene un manipulador de excepciones.
- El control del programa en una excepción lanzada abandona el bloque **try** y busca los bloques **catch**, con el fin de localizar un manipulador apropiado. Si no se lanza una excepción en el bloque **try**, se ignora el manipulador de excepciones para dicho bloque y el programa termina su ejecución después del último bloque **catch**.
- Dentro de un bloque **try**, las excepciones se lanzan en una función o desde una función llamada directa o indirectamente desde el bloque **try**.
- Una vez que se lanza una excepción, el control no se puede devolver directamente al punto de lanzamiento.
- Es posible comunicar información al manipulador de excepciones desde el punto de la excepción. Dicha información es el tipo del objeto arrojado o la información colocada en el objeto arrojado.
- Un popular tipo de excepción arrojada es **char ***. Es común simplemente incluir un mensaje de error como el operando de **throw**.
- Las excepciones arrojadas por una función en particular pueden especificarse por medio de una especificación de excepción. Una especificación de excepción vacía establece que la función no arrojará excepción alguna.
- Las excepciones son capturadas por el manipulador de excepciones más cercano (para el bloque **try** a partir del cual se arrojó la excepción) que especifica el tipo apropiado.
- Como parte del lanzamiento de una excepción, se crea y se inicializa una copia temporal del operando de **throw**. Después, este objeto temporal inicializa la variable apropiada en el manipulador de excepciones. El objeto temporal se destruye cuando abandona el manipulador de excepciones.
- Los errores no siempre se verifican explícitamente. Por ejemplo, un bloque **try** puede aparentemente no contener verificación explícita alguna ni instrucción **throw**. Pero el código al que hace referencia el bloque **try** pudiera, en efecto, provocar la ejecución de código de verificación de errores.
- Una excepción termina el bloque en el que ocurrió la excepción.
- Los manipuladores de excepciones se encuentran contenidos dentro de bloques **catch**. Cada bloque **catch** comienza con la palabra reservada **catch**, seguida por los paréntesis que contienen el tipo y un parámetro opcional para el nombre. Esto es seguido por las llaves que delinean el código para manejo de excepciones. Cuando se captura una excepción, se ejecuta el código en el bloque **catch**. El manipulador **catch** define su propio alcance.
- El parámetro de un manipulador **catch** puede o no tener nombre. Si el parámetro tiene nombre, se puede hacer referencia a él dentro del manipulador. Si el parámetro no tiene nombre (es decir, solamente se lista un tipo con el fin de hacerlo coincidir con el tipo del objeto lanzado, o tres puntos para todos los tipos), entonces el manipulador ignorará al objeto lanzado. El manipulador puede relanzar al objeto hacia un bloque **try** externo.
- Es posible especificar un comportamiento personalizado para reemplazar a la función **terminate** al diseñar otra función para que se ejecute, y proporcionar el nombre de esa función para que se ejecute como el argumento en una llamada a la función **set_terminate**.
- **catch( ... )** significa atrapar todas las excepciones.

- Es posible que ningún manipulador coincida con un objeto lanzado en particular. Esto provoca la búsqueda de una coincidencia para continuar dentro del bloque **try** que lo envuelve.
- Los manipuladores de excepciones se buscan de acuerdo con una coincidencia apropiada. El primer manipulador que arroja una coincidencia se ejecuta. Cuando dicho manipulador termina su ejecución, el control termina con la primera instrucción después del último bloque **catch**.
- El orden de los manipuladores de excepciones afecta la manera en que se manejan las excepciones.
- Un objeto de clase derivada puede capturarse por medio de los manipuladores al especificar el tipo de la clase derivada, o por medio de los manipuladores al especificar los tipos de cualquier clase base de dicha clase derivada.
- Algunas veces, un programa puede procesar muchos tipos de excepciones relacionadas. En lugar de proporcionar clases de excepciones y manipuladores **catch** para cada una, un programador puede proporcionar una excepción y un manipulador **catch** individual para un grupo de excepciones. Al ocurrir cada excepción, el objeto de excepción puede crearse con diferentes datos privados. Este manipulador **catch** puede examinar estos datos privados para distinguir el tipo de la excepción.
- Es posible que aunque exista una coincidencia precisa, se creará una coincidencia que requiera conversiones estándares debido a que el manipulador aparece antes de aquel que resultaría de una coincidencia precisa.
- De manera predeterminada, si no se encuentra un manipulador para una excepción, el programa termina.
- Un manipulador de excepciones no puede acceder directamente a las variables con el alcance de su bloque **try**. La información que necesita el manipulador, por lo general se pasa en el objeto lanzado.
- Los manipuladores de excepciones pueden echar un vistazo más cercano a un error y decidir llamar a **terminate**. Pueden relanzar una excepción. Pueden convertir un tipo de excepción a otro, relanzando una excepción diferente. Pueden realizar cualquier recuperación necesaria y resumir la ejecución después del último manipulador de excepciones. Pueden evaluar la situación que provoca el error, eliminar la causa del error y reintentar llamando a la función original que provocó la excepción (Esto no provocaría una recursividad infinita.) Éstos simplemente pueden devolver algún valor de estado a su entorno, etcétera.
- Un manipulador que atrapa a un objeto de clase derivada debe colocarse antes de que un manipulador atrape un objeto de la clase base. Si un manipulador de la clase base fuera primero, atraparía tanto a los objetos de la clase base como de la clase derivada de dicha clase base.
- Cuando se captura una excepción, es posible que los recursos se hayan almacenado, pero que no se hayan liberado en el bloque **try**. El manipulador **catch** debe liberar estos recursos.
- Es posible que el manipulador **catch** decida que no puede procesar la excepción. En este caso, el manipulador puede simplemente relanzar la excepción. Un **throw** sin argumentos relanza la excepción. Si no se lanzó excepción alguna para comenzar, entonces el relanzamiento provoca una llamada a **terminate**.
- Incluso si un manipulador puede procesar una excepción, y sin importar si hace algún proceso en dicha excepción, el manipulador puede relanzar la excepción para llevar a cabo más procesos fuera del manipulador. Una excepción relanzada se detecta en el siguiente bloque **try** y se manipula mediante un manipulador de excepción listado después del bloque **try** que lo encierra.
- Una función sin especificación de excepciones puede arrojar cualquier excepción.
- La función **unexpected** llama a una función especificada mediante **set_unexpected**. Si no se especifica función alguna de esta manera, se llama de manera predeterminada a **terminate**.
- La función **terminate** puede invocarse de distintas formas: explícitamente; si una excepción arrojada no puede atraparse; si la pila se corrompe durante el manejo de excepciones; como una acción predeterminada durante la llamada a la función **unexpected**; o si, cuando se desenrolla una pila iniciada por una excepción, el destructor intenta arrojar una excepción que provoca la llamada a **terminate**.
- Los prototipos para las funciones **set_terminate** y **set_unexpected** se encuentran en el archivo de encabezado **<exception>**.
- Las funciones **set_terminate** y **set_unexpected** devuelven apuntadores a la última función llamada por **terminate** y por **unexpected**. Esto permite al programador guardar el apuntador a la función, de manera que la pueda recuperar posteriormente.
- Las funciones **set_terminate** y **set_unexpected** toman como argumentos apuntadores hacia funciones. Cada argumento debe apuntar a una función con tipo de retorno **void** y sin argumentos.
- Si la última acción de una función de terminación definida por el usuario no abandona el programa, se llamará a la función **abort** para terminar la ejecución del programa después de la ejecución de las demás instrucciones de terminación de la función definida por el usuario.

- Una excepción arrojada fuera del bloque **try** provocará la terminación del programa.
- Si no se puede localizar un manipulador después de un bloque **try**, la pila continúa desenrollándose hasta que se encuentra un manipulador apropiado. Si finalmente no se localiza el manipulador, entonces se llama a **terminate**, la cual aborta el programa de manera predeterminada con **abort**.
- La especificación de excepciones lista las excepciones que pueden arrojarse desde una función. Una función puede arrojar las excepciones indicadas, o puede arrojar tipos derivados. Si se lanza una excepción no listada en la especificación de excepciones, se llama a **unexpected**.
- Si una función arroja una excepción de un tipo de clase en particular, dicha función también puede arrojar excepciones de todas las clases derivadas de dicha clase con herencia pública.
- Para atrapar una excepción, el manipulador debe tener acceso a un constructor de copia para el objeto arrojado.
- Las excepciones arrojadas desde los constructores provocan que se llame a los destructores para todos los objetos de las clases base completas y los objetos miembro de los objetos que se construyen antes de que se arroje la excepción.
- Si se construyó parcialmente un arreglo de objetos cuando ocurre una excepción, solamente se llamara a los destructores de los objetos construidos por completo en el arreglo de elementos.
- Las excepciones arrojadas desde los destructores pueden atraparse al encerrar a la función que llama al destructor dentro de un bloque **try**, y al proporcionar un manipulador **catch** con el tipo apropiado.
- Una razón poderosa para utilizar la herencia con excepciones es la de crear la habilidad de atrapar fácilmente una variedad de errores relacionados, con una notación concisa. Uno podría atrapar cada tipo de excepción del objeto de una clase derivada de manera individual, pero si todas las excepciones derivadas se manejan igual, es mucho más conciso simplemente atrapar la excepción del objeto de la clase base.
- El C++ estándar especifica que cuando **new** falla, arroja una excepción **bad_alloc** (**bad_alloc** se define en el archivo de encabezado **<new>**).
- Algunos compiladores no cumplen con el estándar de C++, y todavía utilizan la versión de **new** que devuelve **0** ante una falla.
- La función **set_new_handler** (cuyo prototipo se encuentra en el archivo de encabezado **<new>**) toma como argumento un apuntador a una función que no toma argumentos y devuelve **void**. El apuntador a la función se registra como la función a llamar, cuando **new** falla. Una vez que se registra un *manipulador* **new** mediante **set_new_handler**, **new** no arrojará **bad_alloc** ante una falla.
- Un objeto de clase **auto_ptr** mantiene un apuntador hacia la memoria asignada dinámicamente. Cada vez que un objeto **auto_ptr** se sale de alcance, se realiza una operación **delete** en su dato miembro apuntador. La plantilla de clase **auto_ptr** proporciona los operadores ***** y **->**, de modo que el objeto **auto_ptr** puede utilizarse como una variable apuntador normal.
- El estándar de C++ incluye una jerarquía de clases de excepción encabezadas por la clase **exception** (definida en el archivo de encabezado **<exception>**), el cual ofrece el servicio **what()** que se redefine en cada clase derivada para emitir el mensaje de error apropiado.
- Al incluir **std::bad_exception** en la lista de lanzamiento de la definición de una función, si ocurre una excepción inesperada, **unexpected()** arrojará **bad_exception**, en lugar de terminar (de manera predeterminada), o en lugar de llamar a otra función especificada con **set_unexpected**.

## *TERMINOLOGÍA*

**abort()**
aplicación de misión crítica
archivo de encabezado **<exception>**
archivo de encabezado **<memory>**
archivo de encabezado **<new>**
archivo de encabezado **<stdexcept>**
argumento **catch**
atrapar un grupo de excepciones
atrapar una excepción
**auto_ptr**
**bad_alloc**
**bad_cast**
**bad_typeid**
bloque **catch**
bloque envolvente **try**
bloque **try**
**catch(...)**
**catch(void *)**
condición excepcional
declaración de una excepción
desenrollar una pila
**dynamic_cast**
especificación de excepción
especificación de un **throw** vacío
especificación de una excepción vacía
excepción
excepción no atrapada
**exit**
expresión **throw**
función sin especificación de excepciones
**invalid_argument**

lanzamiento sin argumentos
lanzar una excepción
lanzar una excepción inesperada
**length_error**
lista de excepciones
lista de lanzamiento
**logic_error**
macro **assert**
manipulador de excepción
manipulador para una clase base
manipulador para una clase derivada
manipuladores de excepciones anidadas
manipular una excepción
**new_handler**
**nothrow**
objeto de excepción
**out_of_range**
**overflow_error**
punto de lanzamiento
relanzar una excepción
robustez
**runtime_error**
**set_new_handler**
**set_terminate**
**set_unexpected**
**std::bad_exception**
**terminate**
**throw**
tolerancia a fallas
**underflow_error**
**unexpected**

## ERRORES COMUNES DE PROGRAMACIÓN

**23.1** Otra razón por la que las excepciones pueden ser peligrosas como una alternativa al flujo de control normal, es que la pila se desenrolla y los recursos alojados antes de la ocurrencia de la excepción podrían no estar libres. Este problema puede evitarse por medio de una programación cuidadosa.

**23.2** Abandonar un programa puede dejar a un recurso en un estado en el que los demás programas no podrán adquirir dicho recurso; por lo tanto, el programa tendrá una "fuga de recursos".

**23.3** Las excepciones sólo deben lanzarse dentro de un bloque **try**. Una excepción lanzada fuera de un bloque **try** provoca una llamada a la función **terminate**.

**23.4** Es posible lanzar una excepción condicional. Pero tenga cuidado, ya que las reglas de promoción pueden provocar que el valor devuelto por la expresión condicional sea de un tipo diferente al que usted espera. Por ejemplo, cuando se lanza un **int** o un **double** desde la misma expresión condicional, la expresión condicional convertirá el **int** en **double**. Por lo tanto, el resultado siempre será atrapado mediante **catch** con un argumento **double**, en lugar de atraparlo solamente algunas veces como **double** (para el **double** real), y algunas veces atraparlo como **int**.

**23.5** Especificar una lista separada por comas para los argumentos de **catch**, es un error de sitaxis.

**23.6** Colocar **catch( ... )** antes de otros bloques **catch** evita la ejecución de dichos bloques; **catch( ... )** debe colocarse al final de la lista de los manipuladores que siguen al bloque **try**.

**23.7** Colocar un **catch** que atrapa un objeto de una clase base antes de un **catch** que atrapa un objeto de la clase derivada a partir de la clase base, es un error de lógica. El **catch** de la clase base atrapará a todos los objetos de la clase derivada de dicha clase base, por lo que nunca se ejecutará el **catch** de la clase derivada.

**23.8** Colocar un manipulador de excepciones con un argumento de tipo **void *** antes de los manipuladores de excepción con otros tipos de apuntadores, provoca un error de lógica. El manipulador **void** podría atrapar todas las excepciones de los tipos de apuntadores, de modo que los manipuladores nunca se ejecutarían. Solamente **catch ( ... )** puede seguir a **catch( void *)**.

**23.9** Colocar un punto y coma después de un bloque **try** o después de un manipulador **catch** (además del último **catch**) seguido de un bloque **try**, es un error de sintaxis.

**23.10** Asumir que después de procesar una excepción, el control regresará a la primera instrucción después de **throw**, es un error de lógica.

**23.11** Asumir que una excepción lanzada desde un manipulador **catch** se procesará por medio de dicho manipulador o cualquier otro manipulador asociado con el bloque **try** que lanzó la excepción que provocó la ejecución del manipulador **catch** original, es un error lógico.

**23.12** Colocar una instrucción **throw** vacía fuera del manipulador **catch**, y ejecutar dicho manipulador, provoca una llamada a **terminate**.

**23.13** Lanzar una excepción no en la especificación de excepciones de la función, provoca una llamada a **unexpected**.

**23.14** Las clases de excepción definidas por el usuario no necesitan derivarse de la clase **exception**. Por lo tanto, escribir **catch( exception e )** no garantiza el atrapar todas las excepciones que pueda encontrar un programa.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**23.1** Utilice excepciones para errores que deben procesarse en un alcance diferente al que ocurren. Utilice otros medios para manejar los errores que se procesarán en el mismo alcance en el que ocurren.

**23.2** Evite utilizar la manipulación de excepciones para otros propósitos que no sean la manipulación de errores, ya que puede reducir la claridad del programa.

**23.3** Utilice técnicas tradicionales de manejo de errores en lugar de la manipulación de excepciones, para procesar errores locales de manera directa, en donde sea fácil para un programa lidiar con sus propios errores.

**23.4** Asociar cada tipo de error en tiempo de ejecución con el nombre del objeto de excepción apropiado, mejora la claridad del programa.

## *TIPS DE RENDIMIENTO*

**23.1** Aunque es posible utilizar la manipulación de excepciones para propósitos diferentes a la manipulación de errores, esto puede reducir el rendimiento del programa.

**23.2** Por lo general, la manipulación de excepciones se implementa en los compiladores de tal manera que cuando no ocurre una excepción, existe poca o ninguna sobrecarga por la presencia de código de manipulación de excepciones. Cuando ocurren las excepciones, ocurre una sobrecarga en tiempo de ejecución. En realidad, la presencia de código para manipulación de excepciones hace que el programa consuma más memoria.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**23.1** Por lo general, el flujo de control con estructuras de control tradicionales es más claro y más eficiente que con excepciones.

**23.2** El manejo de excepciones se adapta bien en sistemas con componentes desarrollados por separado. El manejo de excepciones facilita la combinación de componentes. Cada componente puede realizar su propia detección de excepciones, de manera separada de la manipulación de excepciones con otro alcance.

**23.3** Cuando trabaje con bibliotecas, es probable que quien llama a la función de la biblioteca tendrá en mente un procesamiento de errores único para una excepción que se genera en la función de la biblioteca. Es poco probable que una función de biblioteca realice el procesamiento de errores que coincida con las necesidades únicas de todos los usuarios. Por lo tanto, las excepciones son un medio apropiado para lidiar con los errores producidos por las funciones de bibliotecas.

**23.4** Una clave para la manipulación de excepciones es que la porción de un programa o sistema que manipulará la excepción puede ser bastante diferente o distante de la porción del programa que detectó y generó la situación excepcional.

**23.5** Si es necesario pasar información acerca del error que provocó la excepción, dicha información puede colocarse en el objeto lanzado. El manipulador **`catch`** contendrá entonces un nombre de parámetro a través del cual se puede hacer referencia a esa información.

**23.6** Un objeto puede lanzarse sin que contenga información a pasar; en este caso, el sólo saber que se lanzó una excepción de este tipo puede proporcionar suficiente información para que el manipulador haga su trabajo correctamente.

**23.7** Una debilidad que se presenta al atrapar excepciones por medio de **`catch( ... )`** es que, por lo general, no se puede asegurar de qué tipo de excepción se trata. Otra debilidad es que sin un parámetro con nombre, no existe forma de hacer referencia al objeto de excepción dentro del manipulador de excepciones.

**23.8** Es mejor incorporar su estrategia de manipulación de excepciones dentro de un sistema, a partir del comienzo del proceso de diseño. Es difícil agregar una manipulación de excepciones efectiva después de que un sistema ya se implementó.

**23.9** Otra razón para no utilizar las excepciones para el flujo de control normal es que estas "excepciones" adicionales pueden interponerse en el camino de las excepciones genuinas de tipos de error. Para el programador se vuelve más difícil dar seguimiento al número de casos de excepciones. Por ejemplo, cuando un programa procesa una variedad excesiva de excepciones, ¿podemos estar seguros de lo que atrapa un **`catch( ... )`**? Las situaciones excepcionales deben ser raras, no comunes.

**23.10** Utilice **`catch( ... )`** para realizar la recuperación que no depende del tipo de excepción, tal como la liberación de recursos comunes. La excepción puede relanzarse para alertar a bloques **`catch`** más específicos.

**23.11** El estándar de C++ recomienda que para hacer programas más robustos, los programadores deben utilizar la versión de **`new`** que lanza excepciones **`bad_alloc`** ante una falla.

**23.12** El objetivo de la jerarquía estándar de **`exception`** es que sirva como un punto de inicio. Los usuarios pueden lanzar excepciones estándar, lanzar excepciones derivadas de las excepciones estándar o lanzar sus propias excepciones no derivadas de las excepciones estándar.

## *TIPS PARA PREVENIR ERRORES*

**23.1** El programador determina el orden en el cual se listan los manipuladores de excepciones. Este orden puede afectar la forma en que se manipulan las excepciones originadas en ese bloque **`try`**. Si usted obtiene un comportamiento inesperado en la manipulación de las excepciones de su programa, podría deberse a que el bloque **`catch`** anterior está interceptando y manipulando las excepciones antes de que alcancen el manipulador que les corresponde.

**23.2** Utilizar la herencia con excepciones permite a un manipulador de excepciones atrapar errores relacionados por medio de dos notaciones más concisas. Ciertamente, una podría atrapar individualmente a cada tipo de apuntador o referencia a una excepción de clase derivada, pero es más conciso atrapar apuntadores o referencias a los objetos de excepción de una clase base. Además, atrapar, uno por uno, apuntadores o referencias a objetos de excepciones de clase derivadas es causa de errores, si el programador olvida probar explícitamente una o más de los tipos de apuntadores o referencias a clases derivadas.

**23.3** Para atrapar todas las excepciones que pudieran arrojarse en un bloque **`try`**, utilice **`catch( ... )`**.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**23.1** Mencione cinco ejemplos comunes de excepciones.

**23.2** Mencione algunas razones por las que no se deben utilizar las técnicas de manipulación de excepciones para el control tradicional del programa.

**23.3** ¿Por qué las excepciones son apropiadas para lidiar con los errores producidos en las funciones de bibliotecas?

**23.4** ¿Qué es una “fuga de recursos”?

**23.5** ¿Si no se arrojan excepciones dentro de un bloque **`try`**, a partir de dónde procede el control, una vez que el bloque **`try`** termina su ejecución?

**23.6** ¿Qué sucede si una excepción se arroja fuera de un bloque **`try`**?

**23.7** Mencione una ventaja clave y una desventaja acerca del uso de **`catch( ... )`**.

**23.8** ¿Qué sucede si ningún manipulador **`catch`** coincide con el tipo del objeto lanzado?

**23.9** ¿Qué sucede si varios manipuladores coinciden con el tipo del objeto lanzado?

**23.10** ¿Por qué un programador especifica un tipo de clase base como el tipo del manipulador **`catch`**, y luego arroja los objetos de tipos de clase derivadas?

**23.11** ¿Cómo se podría escribir un manipulador **`catch`** para procesar tipos relacionados de error, sin utilizar la herencia entre las clases de excepciones?

**23.12** ¿Qué tipo de apuntador se utiliza en un manipulador **`catch`** para atrapar cada excepción de cualquier tipo de apuntador?

**23.13** Suponga que tiene disponible un manipulador **`catch`** con una coincidencia precisa con un tipo de objeto de excepción. ¿Bajo que circunstancias podría ejecutarse un manipulador diferente para los objetos de excepciones para dicho tipo?

**23.14** ¿El lanzamiento de una excepción, debe provocar la terminación del programa?

**23.15** ¿Qué sucede cuando un manipulador **`catch`** lanza una excepción?

**23.16** ¿Qué hace la instrucción **`throw`**?

**23.17** ¿Cómo es que el programador restringe los tipos de excepción que pueden lanzarse desde una función?

**23.18** ¿Qué sucede si una función lanza una excepción de un tipo no permitido por la especificación de la excepción para la función?

**23.19** ¿Qué sucede con los objetos automáticos que se construyeron dentro de un bloque **`try`**, cuando dicho bloque lanza una excepción?

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**23.1** Memoria insuficiente para satisfacer una petición de **`new`**, un subíndice de arreglo fuera de límite, desbordamiento aritmético, división entre cero, parámetros inválidos de una función.

**23.2** (a) La manipulación de excepciones se diseñó para manipular la ocurrencia no frecuente de situaciones que a menudo provocan la terminación del programa, por lo que los creadores de compiladores no están obligados a implementar la manipulación de excepciones para un rendimiento óptimo. (b) El flujo de control mediante estructuras

convencionales de control por lo general es más claro y más eficiente que con las excepciones. (c) Los problemas pueden ocurrir debido a que la pila se desenrolla cuando ocurre una excepción, y los recursos almacenados previos a la excepción podrían no estar liberados. (d) Las excepciones "adicionales" pueden atravesarse en el camino de una excepción genuina de error de tipo. Al programador se le dificulta más dar seguimiento a un número mayor de casos de excepción. ¿Qué es lo que realmente atrapa **catch(. . .)**?

**23.3** Es poco probable que una función de biblioteca realice un procesamiento de error que cumpla con las necesidades específicas de todos los usuarios.

**23.4** Un programa que aborta podría dejar un recurso en un estado en el que otros programas no puedan adquirir dicho recurso.

**23.5** Los manipuladores de excepciones (en los bloques **catch**) para el bloque **try** se ignoran, y el programa termina la ejecución después del bloque **catch**.

**23.6** Una excepción lanzada fuera de un bloque **try** provoca una llamada a **terminate**.

**23.7** La forma **catch(...)** atrapa cualquier tipo de error lanzado dentro de un bloque **try**. Una ventaja es que no escapa ningún error lanzado. Una desventaja es que **catch** no tiene parámetros, de modo que no puede hacer referencia a la información en el objeto lanzado y no puede saber la causa del error.

**23.8** Esto provoca la búsqueda de una coincidencia para continuar en el siguiente bloque **try** envolvente. Al continuar este programa, en algún momento podría determinarse que no existe un manipulador en el programa que coincida con el tipo del objeto lanzado; en este caso se llama a **terminate**, el cual llama a **abort** de manera predeterminada. Se puede proporcionar una alternativa a la función **terminate** como un argumento para **set_terminate**.

**23.9** Se ejecuta el primer manipulador de excepción coincidente después del bloque **try**.

**23.10** Ésta es una buena forma de atrapar tipos relacionados de excepciones.

**23.11** Proporciona una clase de excepción individual y atrapa el manipulador para un grupo de excepciones. Al ocurrir cada excepción, el objeto de excepción puede crearse con diferentes tipos de datos privados. El manipulador **catch** puede examinar estos datos privados para distinguir el tipo de excepción.

**23.12** **void ***.

**23.13** Podría aparecer un manipulador que requiere conversiones estándares, antes que uno con una coincidencia precisa.

**23.14** No, pero termina el bloque en el que se arroja la excepción.

**23.15** La excepción se procesará por medio del manipulador **catch** (si existe alguno) asociado con el bloque **try** (si existe alguno) que encierra al manipulador **catch** que provocó la excepción.

**23.16** Relanza una excepción.

**23.17** Proporciona una especificación de excepción que lista los tipos de excepción que pueden lanzarse desde la función.

**23.18** Llama a la función **unexpected**.

**23.19** A través del proceso de desenrollar una pila, se llama a los destructores para cada uno de estos objetos.

## *EJERCICIOS*

**23.20** ¿Bajo qué circunstancias el programador no proporcionaría un nombre de parámetro cuando define el tipo del objeto que será atrapado por un manipulador?

**23.21** Un programa contiene la instrucción

```
throw;
```

Por lo general, ¿en dónde esperaría encontrar esta instrucción? ¿Qué pasa si dicha instrucción aparece en una parte diferente del programa?

**23.22** Bajo qué circunstancias utilizaría las siguientes instrucciones?

```
catch(...) { throw; }
```

**23.23** Compare la manipulación de excepciones con otros esquemas distintos de manipulación de errores que explicamos en el libro.

**23.24** Liste las ventajas de la manipulación de excepciones con respecto a los métodos convencionales de procesamiento de errores.

**23.25** Utilice la herencia para crear una clase base de excepción y varias clases derivadas de excepciones. Luego, muestre que un manipulador **catch** que especifica la clase base puede atrapar las excepciones de las clases derivadas.

**23.26** Diseñe y escriba un programa para generar y manipular un error de agotamiento de memoria. Su programa debe realizar un ciclo para solicitar la creación de almacenamiento dinámico a través del operador **new**.



# Introducción a las aplicaciones y a los applets de Java

## Objetivos

- Escribir aplicaciones sencillas con Java.
- Utilizar instrucciones de entrada y salida.
- Observar algunas de las excitantes capacidades de Java a través de varios applets de demostración proporcionados con el Java 2 Software Development Kit.
- Comprender la diferencia entre un applet y una aplicación.
- Escribir applets sencillos en Java.
- Escribir archivo sencillos en Lenguaje de Marcación de Hipertexto (HTML) para cargar un applet en el **`applet viewer`** o en un navegador de la World Wide Web.




*Los comentarios son libres, pero los hechos son sagrados.*
C. P. Scott

*El acreedor tiene mejor memoria que el deudor.*
James Howell

*Cuando tengo que tomar una decisión, siempre me pregunto, "¿qué sería lo más divertido?"*
Peggy Walker

*Unas clases fracasan, otras triunfan, y otras son eliminadas.*
Mao Tse Tung

## Plan general



## 24.1 Introducción

Ahora procederemos a estudiar Java, un poderoso lenguaje orientado a objetos, divertido para los novatos, pero también apropiado para los programadores experimentados en la construcción de sistemas de información importantes. Java seguramente será la elección del nuevo milenio para la implementación de aplicaciones basadas en Internet e Intranets, así como el software para dispositivos que se comunican entre redes (tales como teléfonos celulares, paginadores y asistentes digitales personales). ¡No se sorprenda cuando su nuevo estéreo y otros dispositivos en su casa se conecten en red por medio de tecnología Java!

En los capítulos correspondientes a C de este libro presentamos un tratamiento de la programación por procedimientos y el diseño de programas arriba-abajo. En los capítulos de C++, presentamos paradigmas adicionales de programación; programación basada en objetos (con clases, encapsulamiento, objetos, y sobrecarga de operadores), programación orientada a objetos (con herencia y polimorfismo) y programación genérica (con plantillas de funciones y plantillas de clases). Estos paradigmas de programación son cruciales para el desarrollo de sistemas de software elegante, robusto y de fácil mantenimiento. En los capítulos de Java explicamos los gráficos, las interfaces gráficas de usuario, multimedia y la programación orientada a eventos: Sun Microsystems desarrolló Java, teniendo en mente estas populares tecnologías.

Dominar estos variados paradigmas de desarrollo y las tecnologías que explicamos en el libro le ayudará a construir fundamentos sólidos de programación. Trabajamos duro para crear lo que esperamos será una experiencia informativa, entretenida y desafiante para usted.

Una implementación de Java está disponible en el sitio Web de Java

**`java.sun.com`**

Estos capítulos están basados en la versión de Java más reciente de Sun, la *Java 2 Platform*. Sun proporciona una implementación de *Java 2 Platform*, llamada *Java 2 Software Development Kit (J2SDK), versión 1.4* que incluye las herramientas que usted necesita para escribir software en Java. La extraordinaria portabilidad de Java significa que los programas de este libro funcionarán correctamente en cualquier versión de J2SDK 1.4.

En los capítulos 24 a 30, presentamos la programación en Java a una profundidad razonable para un libro introductorio como éste. Usted aprenderá a crear programas en Java llamados aplicaciones y applets; las principales diferencias entre Java, C y C++; la programación orientada y basada en objetos en Java; la programación de gráficos con una variedad de colores, fuentes, contornos de figuras, y formas rellenadas; la programación de interfaces gráficas de usuario (GUIs) con los componentes Swing de Java; y la programación multimedia con efectos tales como clips de audio, procesamiento de imágenes, mapas de imágenes y animación.

## 24.2 Fundamentos de un entorno típico de Java

Por lo general, los sistemas en Java constan de diversas partes: un ambiente, el lenguaje, la interfaz de programación de aplicaciones de Java (API) y varias bibliotecas de clases. La siguiente explicación expone un entorno de programación típico de Java como lo muestra la figura 24.1.

Los programas en Java generalmente pasan a través de cinco fases para poder ejecutarse (figura 24.1). Éstas son: *edición*, *compilación*, *carga*, *verificación* y *ejecución*. Si usted no utiliza UNIX, Windows 95/98 o Windows NT, consulte los manuales para el ambiente Java de su sistema, o pregunte a su profesor cómo llevar a cabo estas tareas en su entorno particular (lo que probablemente será similar al entorno de la figura 24.1).

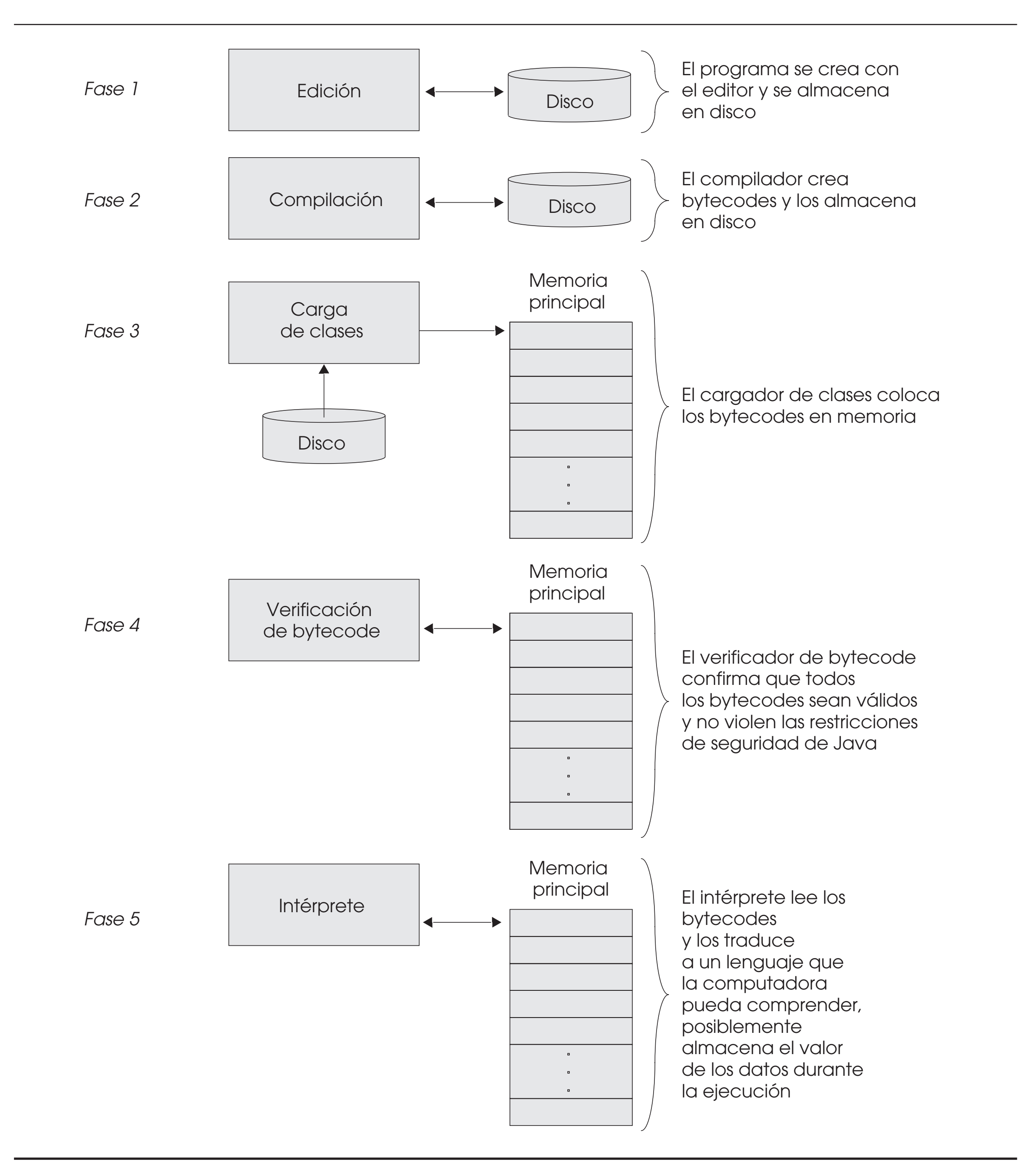



**Figura 24.1** Un típico ambiente de Java.

La fase 1 consiste en editar el archivo. Esto se lleva a cabo con un *programa de edición*. El programador escribe un programa en Java por medio del editor y lo corrige si es necesario. Cuando el programador especifica que debe guardarse el archivo que se encuentra en el editor, el programa se almacena en un dispositivo de memoria secundaria tal como un disco. El archivo con el programa en Java termina con la *extensión* ***.java***. Dos editores ampliamente utilizados en sistemas UNIX son **vi** y emacs. En Windows 95/98 y Windows NT programas de edición sencillos como el comando Edit de MS-DOS y el bloc de notas de Windows serán suficientes. Los ambientes integrados de desarrollo (IDEs) tales como Forte para Java de Sun, JBuilder de Borland, Visual Café de Symantec y Visual J++ de Microsoft tienen editores incluidos que se integran suavemente en el ambiente de programación. Asumimos que el lector sabe cómo editar un archivo.

En la fase 2, el programador aplica el comando ***javac*** para *compilar* el programa. El compilador de Java traduce el programa en Java a *bytecodes*, el lenguaje que comprende el intérprete de Java. Para compilar un programa llamado **Bienvenido.java**, escriba

```
javac Bienvenido.java
```

en la ventana de comando de su sistema (es decir, en el indicador de MS-DOS en Windows 95/98 y Windows NT, o en el indicador del shell en UNIX). Si el programa se compila correctamente, se produce un archivo **Bienvenido.class**. Éste es el archivo que contiene los bytecodes que se interpretarán durante la fase de ejecución.

La fase 3 se llama de *carga*. Esto se hace por medio del *cargador de clases*, el cual toma el archivo (o archivos) **.class** que contiene los bytecodes y los transfiere a la memoria. El archivo **.class** puede cargarse desde un disco en su sistema o sobre una red (tal como la red de su universidad o de su trabajo, o incluso por Internet). Existen dos tipos de programas para los cuales el cargador de clases carga archivos **.class**: *aplicaciones* y *applets*. Una aplicación Java es un programa tal como un procesador de palabras, una hoja de cálculo, un programa de dibujo, un programa de correo electrónico, etcétera, que por lo general se almacena y se ejecuta en memoria desde la computadora local del usuario. Un applet de Java es un pequeño programa que por lo general se almacena en una computadora remota que los usuarios conectan mediante un navegador de la World Wide Web. Los applets se cargan desde una computadora remota en el navegador, se ejecuta en el navegador y se descarta al completar la ejecución. Para ejecutar nuevamente un applet, el usuario debe apuntar su navegador a la ubicación apropiada en la World Wide Web y recargar el programa dentro del navegador.

Las aplicaciones se cargan en memoria y se ejecutan por medio del *intérprete de Java* con el comando ***java***. Cuando se ejecuta una aplicación de Java llamada **Bienvenido**, el comando

```
java Bienvenido
```

invoca al intérprete para la aplicación **Bienvenido** y provoca que el cargador de clases cargue la información utilizada en el programa **Bienvenido**.

El cargador de clases también se ejecuta cuando se carga un applet de Java dentro de un navegador de la World Wide Web como *Netscape Communicator*, *Internet Explorer de Microsoft*, o *HotJava de Sun*. Los navegadores se utilizan para visualizar documentos de la World Wide Web llamados documentos *HTML (Lenguaje de Marcación de Hipertexto)*. El HTML se utiliza para dar formato a un documento, de modo que sea fácil de comprender por la aplicación de navegador (nos introduciremos en HTML en la sección 24.7; para un tratamiento detallado de HTML y otras tecnologías de programación en Internet, revise nuestro libro *Internet and the World Wide Web How to program*). Un documento HTML puede hacer referencia a un applet de Java. Cuando el navegador ve un applet al que se hace referencia dentro de un documento HTML, el navegador lanza el cargador de clases de Java para cargar el applet (por lo general, desde la ubicación en donde se almacena el documento HTML). Los navegadores que soportan Java contienen un intérprete de Java. Una vez que se carga el applet, el intérprete de Java del navegador ejecuta el applet. Además, los applets pueden ejecutarse desde la línea de comando usando el *comando* **appletviewer** proporcionado con el J2SDK; el conjunto de herramientas que incluye el compilador (**javac**), el interprete (**java**), el **appletviewer** y otras herramientas utilizadas por los programadores de Java. Tal como Netscape Communicator, Internet Explorer y HotJava, el **appletviewer** require un documento HTML para invocar un applet. Por ejemplo, si el archivo **Bienvenido.html** hace referencia al applet **Bienvenido**, el comando **appletviewer** se utiliza de la siguiente manera:

```
appletviewer Bienvenido.html
```

Esto provoca que el cargador de clases cargue la información utilizada en el applet **`Bienvenido`**. Por lo general, al **`appletviewer`** se le conoce como un "navegador mínimo"; éste sólo sabe cómo interpretar applets.

Antes de que se ejecuten los bytecodes de un applet por medio del interprete de Java, incluido en un navegador o mediante el **`appletviewer`**, éstos se verifican por medio del *verificador de bytecode* de la fase 4 (esto también sucede con aplicaciones que descargan bytecodes desde una red). Esto garantiza que los bytecodes para la clases que se cargan desde Internet (conocidas como *clases descargables*) sean válidas y que no violen las restricciones de seguridad de Java. Java promueve una fuerte seguridad debido a que los programas en Java que llegan desde una red no deben ser capaces de dañar a sus archivos y a su sistema (como lo hacen los virus).

Por último, en la fase 5, la computadora, bajo el control de su CPU, interpreta el programa un bytecode a la vez, y realiza las acciones especificadas en el programa.

Es posible que los programas no funcionen en el primer intento. Cada una de las fases anteriores puede fallar debido a los diversos errores que explicaremos en este libro. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal en Java, como en la aritmética). Esto provocaría que el programa en Java imprimiera un mensaje de error. El programador regresaría a la fase de edición, haría las correcciones necesarias y procedería de nuevo a través de las fases restantes, para determinar si las correcciones funcionan correctamente.



**Error común de programación 24.1**

*Los errores como la división entre cero ocurren durante la ejecución del programa, de modo que estos errores se llaman errores en tiempo de ejecución o errores de ejecución. Los errores fatales en tiempo de ejecución provocan que los programas terminen de inmediato, sin tener éxito al realizar sus tareas. Los errores no fatales en tiempo de ejecución permiten a los programas completar su ejecución, por lo general con resultados incorrectos.*

La mayoría de los programas en Java introducen datos de entrada y/o salida. Cuando decimos que un programa imprime un resultado, por lo general significa que el resultado se despliega en la pantalla. Los datos pueden emitirse con otros dispositivos tales como discos e impresoras.

## 24.3 Notas generales acerca de Java y de este libro

Java es un lenguaje poderoso. Algunas veces, los programadores experimentados se enorgullecen de su capacidad para usar el lenguaje de manera extraña, contorsionada, y compleja. Ésta es una pobre práctica de programación. Hace que los programas sean más difíciles de leer, más propensos a comportarse de manera extraña, más difíciles de depurar y probar, y más difíciles de adaptar a los requerimientos cambiantes. Estos capítulos también están dedicados a los programadores novatos, de modo que anteponemos la *claridad*. Ésta es nuestra primera "buena práctica de programación".



**Buena práctica de programación 24.1**

*Escriba sus programas en Java de manera sencilla y directa. A esto en ocasiones se le llama KIS ("keep it simple", "manténgalo simple". No deshaga el lenguaje, intentado usos extraños.*

Usted habrá escuchado que Java es un lenguaje portable, y que los programas escritos en Java pueden ejecutarse en muchas computadoras diferentes. *La portabilidad es una meta escurridiza*. El documento del estándar de C contiene una larga lista de temas de portabilidad, y se han escrito libros completos para explicarla.



**Tip de portabilidad 24.1**

*Aunque es más fácil escribir programas portables en Java que en la mayoría de los demás lenguajes de programación, existen diferencias entre los compiladores, los intérpretes y las computadoras que pueden hacer de la portabilidad una meta difícil de alcanzar. El simple hecho de escribir programas en Java, no garantiza la portabilidad. Ocasionalmente el programador necesitará lidiar directamente con las variaciones entre los compiladores y las computadoras.*

**Tip para prevenir errores 24.1**

*Siempre pruebe los programas en Java en todos los sistemas en los que desee ejecutarlos.*

Hicimos un cuidadoso recorrido a través de la documentación de Java de Sun, y comparamos nuestra programación con ésta por motivos de integridad y exactitud. Sin embargo, Java es un lenguaje rico, y existen algunas sutilezas en el lenguaje y algunos temas que no hemos cubierto. Si usted necesita detalles técnicos adicionales, le sugerimos que lea el documento más reciente de Java disponible en Internet y en **`java.sun.com`**.



**Buena práctica de programación 24.2**

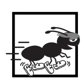
*Lea la documentación para la versión de Java que va a utilizar. Consulte esta documentación con frecuencia para asegurarse de que conoce la rica colección de características de Java y de que utiliza correctamente estas características.*



**Buena práctica de programación 24.3**

*Su computadora y su compilador son buenos maestros. Si después de leer cuidadosamente el manual de la documentación de Java no está seguro de la manera en que funciona una característica de Java, experimente y vea qué sucede. Estudie cada mensaje de error o de advertencia que obtenga cuando compile sus programas, y corríjalos para eliminar dichos mensajes.*

Aquí explicamos cómo funciona Java en su implementación común. Quizá el problema más grave con las primeras versiones de Java es que los programas en Java se ejecutan mediante un intérprete en la máquina del cliente. Los intérpretes se ejecutan muy lento, comparados con los programas totalmente compilados en lenguaje máquina.



**Tip de rendimiento 24.1**

*Los intérpretes tienen una ventaja sobre los compiladores en el mundo de Java, a saber, que un programa interpretado puede comenzar su ejecución de inmediato, tan pronto como se descarga en la máquina del cliente, mientras que un programa a compilarse primero debe sufrir un retraso potencialmente largo mientras el programa se compila antes de que pueda ejecutarse.*

Aunque en los primeros sistemas Java solamente los intérpretes estaban disponibles para ejecutar los bytecodes en el sitio del cliente, los compiladores de Java se escribieron para la mayoría de las plataformas más populares. Estos compiladores toman los bytecodes de Java (o en algunos casos el código fuente de Java) y los compilan en el código de máquina nativo de la máquina del cliente. Estos programas compilados se desempeñan de manera similar al código compilado de C o C++. No existen compiladores para cada plataforma Java, de modo que los programas no podrán ejecutarse al mismo nivel en todas las plataformas.

Los applets presentan algunas características más interesantes. Recuerde, un applet podría provenir virtualmente desde cualquier *servidor Web* del mundo. De modo que el applet tendrá que ser capaz de ejecutarse en cualquier plataforma de Java. En resumen, los applets de rápida ejecución de Java realmente pueden interpretarse. Pero, ¿qué sucede con los applets más grandes y de cómputo intensivo? Aquí, el usuario podría estar dispuesto a sufrir el retraso de la compilación para obtener un mejor rendimiento de ejecución. Para algunos applets especializados de alto rendimiento, el usuario pudiera no tener opción; el código interpretado se ejecutaría muy lentamente para que el código del applet se ejecutara apropiadamente, por lo que el applet tendría que compilarse.

Un paso intermedio entre los intérpretes y los compiladores es un *compilador justo a tiempo (JIT, just in time)* que, mientras se ejecuta el compilador, produce código compilado para los programas y los ejecuta en lenguaje máquina, en lugar de reinterpretarlos. Los compiladores JIT no producen código máquina, que es tan eficiente como un compilador completo. En la actualidad, los compiladores completos para Java se encuentran en desarrollo. Para obtener la información más reciente sobre la traducción de un programa en Java a alta velocidad, puede leer acerca del compilador *HotSpot* de Sun, visite

**`java.sun.com/products/hotspot/`**

Para las empresas que quieren desarrollar sistemas de información de trabajo pesado, los ambientes integrados de desarrollo (IDEs) de las empresas de software más importantes están disponibles. Los IDEs proporcionan muchas herramientas para soportar el proceso de desarrollo de software. Actualmente, muchos IDEs de Java en el mercado son tan poderosos como aquellos disponibles para el desarrollo de sistemas en C y C++. Ésta es una clara señal de que Java ya ha sido aceptado como un lenguaje viable para el desarrollo de importantes sistemas de software.

## 24.4 Un programa sencillo: Impresión de una línea de texto

Comenzaremos considerando una sencilla *aplicación* en Java que despliega una línea de texto. Una aplicación es un programa que se ejecuta por medio del intérprete **`java`** (el cual explicaremos más adelante en esta sección). El programa y su salida aparecen en la figura 24.2.

Este programa muestra varias características importantes del lenguaje Java. Consideraremos con detalle cada línea del programa. Cada programa tiene las líneas numeradas, para conveniencia del lector; dichos números de línea no son parte de los programas Java. La línea 7 hace el "trabajo real" del programa, a saber, despliega en la pantalla la frase **`Bienvenido a la programacion en Java!`**. Pero, consideremos cada línea en orden. La línea 1

```
// Figura 24.2: Bienvenido1.java
```

comienza con **`//`**, lo que indica que el resto de la línea es un *comentario*. Comenzamos cada programa con un comentario que indica el número y el nombre del archivo. Como en C++, a un comentario que comienza con **`//`** se le llama *comentario de una sola línea*, debido a que el comentario termina al final de la línea actual.

Java también soporta comentarios de varias líneas (delimitados con **`/*`** y **`*/`**), los cuales presentamos en el capítulo 2; una manera similar de hacer comentarios, llamada *comentario para documentación*, se delimita con **`/**`** y **`*/`**.



**Error común de programación 24.2**

*Olvidar uno de los delimitadores de un comentario de varias líneas, es un error de sintaxis.*

Por lo general, los programadores en Java utilizan comentarios de una sola línea al estilo C++, con más preferencia que los comentarios al estilo C. A través de este libro, utilizaremos comentarios de una sola línea al estilo C++. Java introdujo la sintaxis del comentario de documentación para permitir a los programadores resaltar porciones de programas, que el programa de utilidad **`javadoc`** (proporcionado por Sun Microsystems con el Java 2 Software Development Kit) pueda leer y utilizar para preparar automáticamente la documentación de sus sistemas. Existen algunos aspectos sutiles para utilizar adecuadamente los comentarios estilo **`javadoc`** dentro de un programa. En este libro no utilizaremos los comentarios al estilo **`javadoc`**.

La línea 4

```
public class Bienvenido1 {
```

comienza la *definición de la clase* **`Bienvenido1`**. Cada programa en Java consta de al menos una definición de clase definida por usted, el programador. A estas clases se les conoce como *clases definidas por el programador* o *clases definidas por el usuario*. En el capítulo 26, explicamos programas que contienen varias clases definidas por el programador. La *palabra reservada* ***`class`*** introduce la definición de una clase en Java y va inmediatamente seguida por el *nombre de la clase* (**`Bienvenido1`** en este programa). Las palabras reservadas (o palabras clave) se reservan para el uso de Java (a lo largo del libro explicamos las palabras reservadas), y siempre se escriben con letras minúsculas. Por convención, todos los nombres de las clases en Java comienzan

```
1  // Figura 24.2: Bienvenido1.java
2  // Primer programa en Java
3
4  public class Bienvenido1 {
5     public static void main( String args[] )
6     {
7        System.out.println( "Bienvenido a la programacion en Java!" );
8     } // fin de main
9  } // fin de la clase Bienvenido1
```

```
Bienvenido a la programacion en Java!
```

**Figura 24.2** Primer programa en Java.

con una letra mayúscula, y tienen una letra mayúscula por cada palabra en el nombre de la clase (por ejemplo, **NombreClaseEjemplo**). Al nombre de la clase se le llama *identificador*. Un identificador es una serie de caracteres que consta de letras, dígitos, guiones bajos (_) y símbolos de moneda (**$**), que no comienzan con un dígito y no contienen espacios. Algunos identificadores válidos son **Bienvenido1**, **$valor**, **_valor**, **m_campoEntrada1**, y **boton7**. El nombre **7boton** no es un identificador válido debido a que comienza con un dígito, y el nombre **campo entrada** no es un identificador válido debido a que contiene un espacio. Java es *sensible a mayúsculas y minúsculas*, las letras mayúsculas y minúsculas son diferentes, de modo que **a1** y **A1** son identificadores diferentes.



**Error común de programación 24.3**

*Java es sensible a mayúsculas y minúsculas. Por lo general, no utilizar las letras mayúsculas y minúsculas apropiadas para un identificador, es un error de sintaxis.*



**Buena práctica de programación 24.4**

*Por convención, usted siempre debe comenzar el nombre de una clase con la primera letra en mayúscula.*



**Buena práctica de programación 24.5**

*Cuando lea un programa en Java, busque identificadores que comiencen con la primera letra en mayúscula. Por lo general, éstos representan clases de Java.*



**Observación de ingeniería de software 24.1**

*Evite utilizar identificadores que contengan signos de moneda (**$**), ya que con frecuencia el compilador los utiliza para crear nombres de identificadores.*

En los capítulos 24 y 25, toda clase que definimos comienza con la *palabra reservada* **public**. Por ahora, solamente requeriremos esta palabra reservada. En el capítulo 26, explicaremos con detalle la palabra reservada **public**, y también explicaremos las clases que no comienzan con dicha palabra reservada. [*Nota:* En este libro, muchas veces le pedimos que simplemente imitara ciertas características de Java que presentábamos mientras usted escribía sus propios programas en Java. Esto lo hacemos específicamente cuando aún no es importante conocer todos los detalles acerca de una característica de Java. De inicio, todos los programadores aprenden cómo programar, imitando lo que otros programadores han hecho antes que ellos. En cada detalle que le pedimos que imite, le indicamos en dónde se encuentra la explicación completa que le daremos más adelante.]

Cuando usted guarda la definición de una clase dentro de un archivo, el nombre de la clase debe utilizarse como parte del nombre del archivo. Para nuestras aplicaciones, el nombre del archivo es **Bienvenido1.java**. Todas las definiciones de clases en Java se almacenan en archivos que terminan con la extensión de archivo **.java**.



**Error común de programación 24.4**

*Para una clase pública, es un error si el nombre de archivo no es idéntico al nombre de la clase tanto en las letras, como en las mayúsculas y las minúsculas. Por lo tanto, también es un error que un archivo contenga dos o más clases públicas.*



**Error común de programación 24.5**

*Es un error no finalizar el nombre de un archivo con la extensión **.java**, si contiene la definición una clase de la aplicación. El compilador de Java no podrá compilar la definición de la clase.*

Una *llave izquierda* (al final de la línea 4 de este programa), **{**, comienza el *cuerpo* de cada definición de clase. Observe que las líneas 5 a 8 están sangradas. Ésta es una convención de espaciado utilizada para hacer más legibles los programas. Definimos cada convención de espaciado como una b*uena práctica de programación*.



**Error común de programación 24.6**

*Si las llaves no están en pares coincidentes, el compilador indica un error.*



**Buena práctica de programación 24.6**

*Cada vez que introduzca una llave izquierda de apertura,* **{**, *en su programa, introduzca inmediatamente la llave derecha de cierre,* **}**, *y vuelva a colocar el indicador entre las llaves para comenzar a introducir el cuerpo del programa. Esto ayuda a evitar que falten llaves.*



**Buena práctica de programación 24.7**

*Sangre el cuerpo entero de cada definición de clase un "nivel" entre la llave izquierda,* **{**, *y la llave derecha,* **}**, *que define el cuerpo de la clase. Esto enfatiza la estructura de la definición de la clase, y ayuda a que las definiciones de clases sean más fáciles de leer.*



**Buena práctica de programación 24.8**

*Establezca una convención para el tamaño del sangrado que prefiera, y entonces aplique de manera uniforme dicha convención. Puede utilizar la tecla tab para crear el sangrado, aunque tab podría variar entre editores. Le recomendamos el uso de tabuladores de 1/4 de pulgada o (preferiblemente) tres espacios para formar un nivel de sangrado.*

La línea 5

```
public static void main( String args[] )
```

es parte de cada aplicación en Java. Las aplicaciones en Java comienzan automáticamente en **main**. Los paréntesis después de **main** indican que **main** es un *método* de programa, o lo que un programador en C o C++ llamaría una función. Por lo general, las definiciones de clases en Java contienen uno o más métodos. Para una clase de aplicación en Java, exactamente uno de esos métodos debe llamarse **main** y debe definirse como muestra la línea 5; de lo contrario, el intérprete de java no ejecutará la aplicación. Los métodos son capaces de realizar tareas y devolver información cuando llevan a cabo sus funciones. La *palabra reservada* ***void*** indica que este método realizará una tarea (en este programa despliega una línea de texto), pero no devolverá información alguna cuando complete su tarea. Veremos que muchos métodos devuelven información cuando completan su tarea. En el capítulo 25 explicaremos con detalle los métodos. Por ahora, simplemente imite la primera línea en cada una de las aplicaciones en Java.

La llave izquierda, **{**, de la línea 6 comienza el *cuerpo de la definición del método*. Su correspondiente llave derecha, **}**, debe terminar el cuerpo de la definición del método (línea 8 del programa). Observe que la línea en el cuerpo del método se sangra entre las dos llaves.



**Buena práctica de programación 24.9**

*Sangre por completo el cuerpo de cada definición de método un "nivel" entre la llave izquierda,* **{**, *y la llave derecha,* **}**. *Esto hace que la estructura del método resalte, y ayuda a que la definición del método sea más fácil de leer.*

La línea 7

```
System.out.println( "Bienvenido a la programacion en Java!" );
```

instruye a la computadora para que imprima la *cadena* de caracteres contenida entre las comillas dobles. En ocasiones, a una cadena se le llama *cadena de caracteres*, un *mensaje* o una *literal de cadena*. Por lo general, nos referiremos a los caracteres entre comillas como cadenas. El compilador no ignora los caracteres blancos en una cadena.

A **system.out** se le conoce como *objeto estándar de salida*. **System.out** permite a las aplicaciones en Java desplegar cadenas y otro tipo de información en la *ventana de comando* desde la que se ejecuta la aplicación en Java. En Windows 95/98 de Microsoft, la ventana de comando es el *indicador* de *MS-DOS*. En Windows NT de Microsoft, la ventana de comando es el *Indicador de comandos*. En UNIX, a la ventana de comando por lo general se le llama *ventana de comando*, *herramienta shell* o *shell*. En computadoras que ejecutan un sistema operativo que no tiene una ventana de comando (tal como Macintosh), el intérprete **java** por lo general despliega una ventana que contiene la información que despliega el programa.

El método ***System.out.println*** *despliega* (o *imprime*) una *línea* de texto en la ventana de comando. Cuando **System.out.println** completa su tarea, coloca el *cursor de salida* (la ubicación en donde se desplegará el siguiente carácter) al principio de la siguiente línea en la ventana de comando (esto es similar a opri-

mir la tecla *Entrar,* cuando escribe en un editor de texto; el cursor se reposiciona al principio de la siguiente línea de su archivo).

A la línea completa, incluido **`System.out.println`**, su *argumento* en los paréntesis (la cadena) y el *punto y coma* (**`;`**), se le llama *instrucción*. Toda instrucción debe terminar con un punto y coma (también llamado *terminador de instrucción*). Cuando se ejecuta esta instrucción, se despliega el mensaje **`Bienvenido a la programacion en Java!`** en la ventana de comando.

**Error común de programación 24.7**

*Omitir el punto y coma al final de una instrucción, es un error de sintaxis.*



**Tip para prevenir errores 24.2**

*Cuando el compilador reporta un error de sintaxis, el error podría no estar en la línea que indica el mensaje de error. Primero, verifique la línea en donde se reporta el error. Si la línea no contiene errores de sintaxis, verifique las líneas anteriores del programa.*

Ahora estamos listos para compilar y ejecutar nuestro programa. Para compilar el programa, abrimos la ventana de comando, nos cambiamos al directorio en donde se encuentra almacenado el programa y escribimos

```
javac Bienvenido1.java
```

Si el programa no contiene errores de sintaxis, el comando anterior crea un nuevo archivo llamado **`Bienvenido1.class`** que contiene los bytecodes de Java que representan a nuestra aplicación. Estos bytecodes serán interpretados por el intérprete **`java`** cuando le indiquemos que ejecute el programa al escribir el comando

```
java Bienvenido1
```

el cual inicia el intérprete e indica que debe cargarse el archivo **`.class`** para la clase **`Bienvenido1`**. Observe que se omite la extensión **`.class`** del nombre del archivo del comando anterior; de lo contrario, el intérprete no ejecutará el programa. El intérprete llama automáticamente al método **`main`**. A continuación, la instrucción de la línea 7 de **`main`** despliega "**`Bienvenido a la programacion en Java!`**". La figura 24.3 muestra la ejecución de la aplicación en la ventana de MS-DOS.

Aunque este primer programa despliega la salida en la ventana de comandos, la mayoría de las aplicaciones Java que despliegan la salida utilizan ventanas o *cuadros de diálogo*. Por ejemplo, los navegadores de la World Wide Web, tales como Netscape Communicator o Microsoft Internet Explorer despliegan las páginas Web en sus propias ventanas. Por lo general, los programas de correo electrónico le permiten escribir un mensaje en una ventana proporcionada por el programa de correo electrónico, o leer los mensajes que recibe en una ventana proporcionada por el programa de correo electrónico. Los cuadros de diálogo son ventanas que por lo general se utilizan para desplegar mesajes importantes para el usuario de una aplicación. Java 2 ya incluye la clase ***`JOPtionPane`*** que le permiten desplegar fácilmente un cuadro de diálogo que contiene información. El programa de la figura 24.4 despliega una cadena similar a la que aparece en la figura 24.2 en un cuadro de diálogo predefinido llamado *diálogo de mensaje*. Observe que esta nueva versión del programa también utiliza la *secuencia de escape* al estilo C, **`\n`**, para insertar en la cadena caracteres de nueva línea.

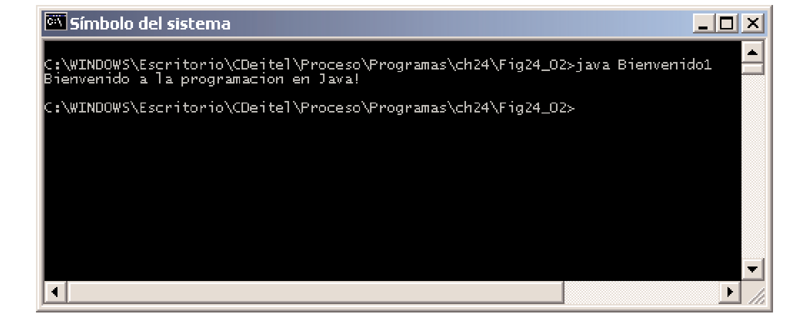



**Figura 24.3** Ejecución de la aplicación **`Bienvenido1`** en la ventana de MS-DOS (Símbolo del sistema).

```
1  // Figura 24.4: Bienvenido2.java
2  // Impresión de múltiples líneas en un cuadro de diálogo
3  import javax.swing.JOptionPane;  // importa la clase JOptionPane
4
5  public class Bienvenido2 {
6     public static void main( String args[] )
7     {
8        JOptionPane.showMessageDialog(
9           null, "Bienvenido\na la\nprogramacion\nen Java!" );
10
11       System.exit( 0 );  // termina el programa
12    } // fin de main
13 } // fin de la clase Bienvenido2
```

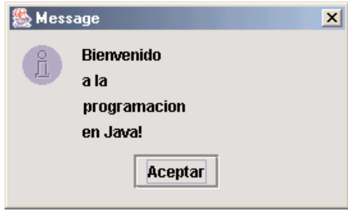



**Figura 24.4** Cómo desplegar varias líneas en un cuadro de diálogo.

Una de las grandes fortalezas de Java es su rica colección de clases predefinidas, las cuales pueden reutilizar los programadores en lugar de "reinventar la rueda". En el libro, utilizamos un gran número de estas clases. Las diversas clases predefinidas se agrupan en categorías de clases relacionadas llamadas *paquetes*. A los paquetes se les conoce de manera colectiva como *biblioteca de clases de Java* o como la *interfaz de programación de aplicaciones de Java (API)*. La clase **JOptionPane** está definida para nosotros en un paquete llamado ***javax.swing***.

La línea 3

```
import javax.swing.JoptionPane;
```

es una instrucción para *importar*. El compilador utiliza instrucciones **import** para identificar y cargar las clases requeridas para compilar un programa en Java. Cuando usted utiliza clases de la API de Java, el compilador intenta garantizar que usted las utiliza correctamente. Las instrucciones **import** ayudan al compilador a localizar las clases que intenta utilizar. Cada porción del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en la API de Java se almacenan en el directorio **java** o **javax** que contiene muchos subdirectorios, incluso **swing** (un subdirectorio de **javax**). En el capítulo 26, explicaremos con detalle los paquetes.

La línea anterior le indica al compilador que cargue la clase ***JOptionPane*** desde el paquete ***javax.swing***. Este paquete contiene muchas clases que ayudan a los programadores en Java a definir un interfaces gráficas de usuario (GUI) para sus aplicaciones. Los *componentes GUI* facilitan la entrada de datos por parte del usuario de su programa, y el dar formato o presentar la salida de datos para el usuario de su programa. Por ejemplo, la figura 24.5 contiene una ventana de Microsoft Internet Explorer. En la ventana, existe una barra que contiene *menús* (**Archivo**, **Edición**, **Ver**, etcétera). Bajo la barra de menú hay un conjunto de *botones*, los cuales tienen una tarea definida dentro del Microsoft Internet Explorer. Debajo de los botones existe un *campo de texto* en el que el usuario puede introducir el nombre del sitio a visitar dentro de la World Wide Web. A la izquierda del campo de texto existe una *etiqueta* que indica el propósito del campo de texto. Los menús, botones, campos de texto y etiquetas forman parte del GUI del Microsoft Internet Explorer. Todos ellos le permiten a usted interactuar con el programa Explorer. Java contiene clases que implementan los componentes de la GUI descritas aquí, y otras que describiremos en el capítulo 29. En **main**, las líneas 8 y 9

```
JOptionPane.showMessageDialog(
   null, "Bienvenido\na la programacion\nen Java!" );
```

indican una llamada al método **`showMessageDialog`** de la clase **`JOptionPane`**. El método requiere dos argumentos. Cuando un método requiere varios argumentos, éstos se separan con *comas* (**`,`**). Hasta que expliquemos **`JOptionPane`** con detalle en el capítulo 29, el primer argumento siempre será la palabra reservada ***`null`***. El segundo argumento es la cadena a desplegar.



**Buena práctica de programación 24.10**

*Coloque un espacio después de cada coma (`,`) en una lista de argumentos, para hacer más legibles los programas.*

El método **`JOptionPane.showMessageDialog`** es un método de la clase **`JOptionPane`** llamado *método estático*. Dichos métodos siempre se llaman mediante el nombre de la clase, seguido por un operador punto (**`.`**) y el nombre del método. En el capítulo 26, explicaremos los métodos estáticos.

Al ejecutar la instrucción anterior se despliega el cuadro de diálogo que muestra la figura 24.6. La *barra de título* del diálogo contiene la cadena **Message** para indicar que el diálogo presenta un mensaje para el usuario. El cuadro de diálogo incluye automáticamente el botón **ACEPTAR** que permite al usuario utilizarlo para *retirar (ocultar) el diálogo* al presionar el botón. Esto se lleva a cabo colocando el *cursor del ratón* (también llamado *apuntador del ratón*) sobre el botón **ACEPTAR** y haciendo clic con el ratón.

Recuerde que todas las instrucciones en Java terminan con un punto y coma (**`;`**). Por lo tanto, las líneas 8 y 9 representan una instrucción. Java permite a instrucciones grandes dividirse en varias líneas. Sin embargo, usted no puede dividir una instrucción en medio de un identificador o en el centro de la cadena.



**Error común de programación 24.8**

*Dividir una instrucción a la mitad de un identificador o de una cadena, es un error de sintaxis.*

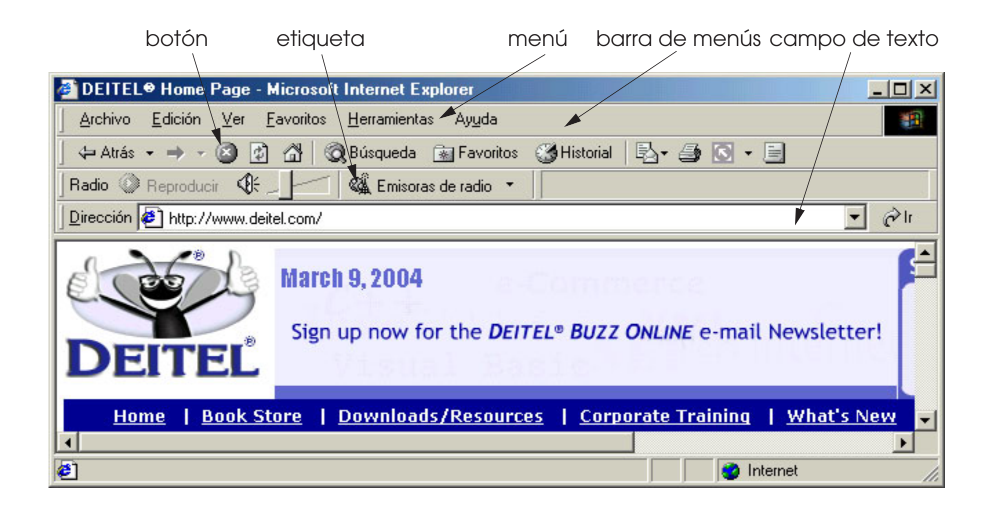



**Figura 24.5** Ventana de Microsoft Internet Explorer con componentes GUI.




**Figura 24.6** Diálogo de mensaje.

La línea 11

```
System.exit( 0 );   // termina el programa
```

utiliza el método estático ***exit*** de la clase **System** para terminar la aplicación. Esta línea es necesaria en cualquier aplicación que despliegue una interfaz gráfica de usuario, para terminar la aplicación. De nuevo, observe la sintaxis utilizada para llamar al método, el nombre de la clase (**System**), un punto (**.**) y un nombre de método (**exit**). Recuerde que los identificadores que comienzan con una letra mayúscula, por lo general representan nombres de clases. Por lo tanto, usted puede asumir que **System** es una clase. El argumento **0** para el método **exit** indica que las aplicaciones terminaron exitosamente (por lo general un valor diferente de cero indica que ocurrió un error). Este valor se pasa a la ventana de comando que ejecuta el programa. Esto es útil si el programa se ejecuta desde un archivo batch (en sistemas Windows 95/98/NT), o mediante un script del shell (en sistemas UNIX). Por lo general, los archivos batch y los scripts se utilizan para ejecutar programas en secuencia, de manera que cuando termina el primer programa, comienza automáticamente la ejecución del siguiente programa. Para mayor información acerca de los archivos batch o los scripts del shell, revise la documentación de su sistema operativo.

La clase **System** es parte del paquete ***java.lang***. Observe que la clase **System** no se importa mediante una instrucción **import** al principio del programa. El paquete ***java.lang*** se importa automáticamente en cada programa en Java.



**Error común de programación 24.9**

*Olvidar llamar a* ***System.exit*** *en una aplicación que despliega una interfaz gráfica, evita que el programa termine de manera apropiada. Por lo general, esto provoca que no sea posible introducir comando alguno.*

## 24.5 Otra aplicación en Java: Suma de enteros

Nuestra siguiente aplicación introduce dos enteros (números completos) escritos por el usuario desde el teclado, calcula la suma de estos valores y despliega el resultado. Conforme el usuario introduce cada entero y presiona la tecla *Entrar*, el entero se introduce en el programa y se suma al total.

Este programa utiliza otro cuadro de diálogo predefinido desde la clase **JOPtionPane** llamado *diálogo de entrada*, el cual permite al usuario introducir un valor a utilizarse en el programa. El programa también utiliza un diálogo de mensaje para desplegar los resultados de la suma. La figura 24.7 muestra la aplicación y las capturas de las pantallas de prueba.

```
// Figura 24.7: Suma.java
// Un programa de suma

import javax.swing.JOptionPane;  // importa la clase JOptionPane

public class Suma {
   public static void main( String args[] )
   {
      String primerNumero,    // primera cadena introducida por el usuario
             segundoNumero;   // segunda cadena introducida por el usuario
      int numero1,            // primer número a sumar
          numero2,            // segundo número a sumar
          suma;               // suma de numero1 y numero2

      // lee el primer número del usuario como una cadena
      primerNumero =
         JOptionPane.showInputDialog( “Introduzca el primer entero” );

      // lee el segundo número del usuario como una cadena
```

**Figura 24.7** Un programa de suma “en acción”. (Parte 1 de 2.)

```
20         segundoNumero =
21            JOptionPane.showInputDialog( "Introduzca el segundo entero" );
22
23         // convierte los números del tipo String a tipo int
24         numero1 = Integer.parseInt( primerNumero );
25         numero2 = Integer.parseInt( segundoNumero );
26
27         // suma los números
28         suma = numero1 + numero2;
29
30         // despliega los resultados
31         JOptionPane.showMessageDialog(
32            null, "La suma es " + suma, "Resultados",
33            JOptionPane.PLAIN_MESSAGE );
34
35         System.exit( 0 );   // termina el programa
36      } // end main
37   } // fin de la clase Suma
```

**Figura 24.7** Un programa de suma "en acción". (Parte 2 de 2.)

La línea 4

```
import javax.swing.JOptionPane;   // importa la clase JOptionPane
```

especifica al compilador en dónde localizar **`JOptionPane`** para utilizarlo con esta aplicación.

Como lo establecimos previamente, todo programa en Java consta de al menos una definición de clase. La línea 6

```
public class Suma {
```

comienza las definiciones de la clase **`Suma`**. El nombre de archivo para esta clase pública debe ser **`Suma.java`**.

Recuerde que todas las definiciones de clases comienzan con una llave izquierda de apertura (final de la línea 6), **`{`**, y con una llave derecha de cierre, **`}`** (línea 37).

Como establecimos anteriormente, toda aplicación comienza su ejecución con el método **`main`** (línea 7). La llave izquierda (línea 8) marca el inicio del cuerpo de **`main`** y su correspondiente llave izquierda (línea 36) marca el final.

Las líneas 9 y 10

```
String primerNumero,     // primera cadena introducida por el usuario
       segundoNumero     // segunda cadena introducida por el usuario
```

forman una *declaración*. Las palabras **`primerNumero`** y **`segundoNumero`** son los nombres de las *variables*. Todas las variables deben declararse con el nombre y el tipo de dato, antes de que puedan utilizarse dentro de

un programa. Esta declaración especifica que las variables **primerNumero** y **segundoNumero** son tipos de datos **String** (del paquete **java.lang**), lo cual significa que estas variables almacenarán cadenas. Un nombre de variable puede ser cualquier identificador válido. Las declaraciones terminan con punto y coma (**;**), y pueden dividirse en varias líneas, con cada variable en la declaración separada por una coma (es decir, una *lista separada por comas* de nombres de variables). Es posible declarar varias variables en una declaración o en declaraciones múltiples. Podríamos haber escrito dos declaraciones, una para cada variable, pero la declaración anterior es más concisa. Observe los comentarios de una sola línea al final de cada línea. Ésta es una sintaxis común utilizada por los programadores para indicar el propósito de cada variable en el programa.



### Buena práctica de programación 24.11

*Elegir nombres de variables significativas (descriptivas) ayuda a un programa a estar "autodocumentado" (es decir, se vuelve más sencillo comprender un programa sólo con leerlo, y no es necesario tener que leer los manuales o utilizar comentarios en exceso).*



### Buena práctica de programación 24.12

*Por convención, los identificadores de nombres de variables comienzan con una letra minúscula. Así como con los nombres de las clases, cada palabra del nombre después de la primera, debe comenzar con una letra mayúscula. Por ejemplo, el identificador **primerNumero** tiene una letra mayúscula **N** en la segunda palabra **Numero**.*



### Buena práctica de programación 24.13

*Algunos programadores prefieren declarar cada variable en una línea aparte. Este formato permite insertar fácilmente un comentario descriptivo después de cada declaración.*

En las líneas 11 a 13

```
int numero1,      // primer número a sumar
    numero2,      // segundo número a sumar
    suma;         // suma el numero1 y el numero2
```

declaran que las variables **numero1**, **numero2** y **suma** son datos de tipo **int**.

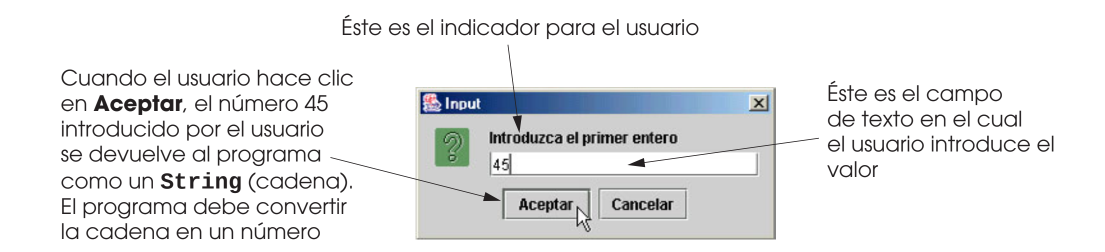
Pronto explicaremos los tipos de datos **float** y **double** para especificar números reales y variables de tipo **char** para especificar datos de tipo carácter. Una variable **char** puede contener solamente una letra minúscula, una letra mayúscula, un solo dígito, o un carácter especial tal como **×**, **$**, **7**, ***** y secuencias de escape (tales como el carácter de nueva línea **\n**). Además, Java es capaz de representar caracteres de muchos otros idiomas.

Con frecuencia, a los tipos tales como **int**, **double** y **char** se les llama *tipos de datos primitivos* o *tipos de datos predefinidos*. Los nombres de los tipos primitivos son palabras reservadas. En el capítulo 25 resumimos los ocho tipos de datos primitivos (**bolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**).

Las líneas 15 a 17

```
// lee el primer número del usuario como una cadena
primerNumero =
   JOptionPane.showInputDialog ( "Introduzca el primer entero" );
```

lee un **String** introducido por el usuario que representa el primero de dos enteros que se sumarán. El método **JOptionPane.showInputDialog** despliega el siguiente cuadro de diálogo:



El argumento de **showInputDialog** indica al usuario qué hacer en el siguiente campo. A este mensaje se le llama *indicador* debido a que le señala al usuario que realice una acción específica. El usuario escribe caracteres en el campo de texto, luego hace clic en el botón **Aceptar** para devolver la cadena al programa. [Si usted escribe y nada aparece en el campo de texto, coloque el apuntador del ratón en el campo de texto y haga clic con el ratón para activar dicho campo.] Desafortunadamente Java no proporciona una forma sencilla de entrada que sea análoga, para desplegar una salida en la ventana de comandos con **System.out.print** y **System.println**. Por esta razón, normalmente recibimos la entrada de un usuario a través de un componente GUI (un diálogo de entrada en este programa).

Técnicamente el usuario puede escribir cualquier cosa en el campo de texto de entrada. En este programa, si el usuario digita un valor no entero o hace clic en el botón **Cancelar**, ocurrirá un error lógico en tiempo de ejecución.

El resultado de llamar a **JOptionPane.showInputDialog** (un **String** que contiene los caracteres digitados por el usuario) se da a la variable **primerNumero** con el *operador de asignación* **=**. La instrucción se lee como, "**primerNumero** *obtiene* el valor **JOptionPane.ShowInutDialog( "Introduzca el primer entero" )**". El operador **=** es un *operador binario* debido a que tiene dos *operandos*: **primerNumero** y el resultado de la expresión **JOptionPane.showInputDialog( "Introduzca el primer entero" )**. A esta instrucción completa se le llama *instrucción de asignación*, debido a que asigna un valor a una variable. La expresión al lado derecho del operador de asignación **=**, siempre se avalúa primero.

Las líneas 19 a 21

```
// lee el segundo número del usuario como una cadena
segundoNumero =
   JOptionPane.showInputDialog( "Introduzca el segundo entero " );
```

despliegan un diálogo de entrada en el que el usuario escribe un **String** que representa el segundo de los dos enteros que se sumarán.

Las líneas 23 a 25

```
// convierte los números del tipo String a tipo int
numero1 = Integer.parseInt( primerNumero );
numero2 = Integer.parseInt( segundoNumero );
```

convierten dos cadenas introducidas por el usuario a valores **int** que pueden utilizarse en el cálculo. El método **Integer.parseInt** (un método estático de la clase **Integer**) convierte su argumento de tipo **String** a un entero. La clase **Integer** es parte del paquete **java.lang**. El entero devuelto por **Integer.parseInt** de la línea 24 se asigna a la variable **numero1**. Cualquier referencia subsiguiente a **numero1** en el programa utiliza el mismo valor entero. El entero devuelto por **Integer.parseInt** en la línea 25 se asigna a la variable **numero2**. Cualquier referencia subsiguiente a **numero2** en el programa utiliza el mismo valor entero.

La instrucción de asignación de la línea 28

```
suma = numero1 + numero2;
```

calcula la suma de las variables **numero1** y **numero2**, y asigna el resultado a la variable **suma** mediante el uso del operador de asignación **=**. La instrucción se lee como, "**suma** *obtiene* el valor de **numero1** + **numero2**". La mayoría de los cálculos se realiza en instrucciones de asignación.



**Buena práctica de programación 24.14**

*Coloque espacios de cualquier lado de un operador binario. Esto hace que el operador sobresalga y hace al programa más legible.*

Después de realizar los cálculos, en las líneas 31 a 33

```
JOptionPane.showMessageDialog(
   null, "La suma es " + suma, "Resultados",
   JOptionPane.PLAIN_MESSAGE );
```

utilizan el método **JOptionPane.showMessageDialog** para desplegar el resultado de la suma. La expresión

```
"La suma es " + suma
```

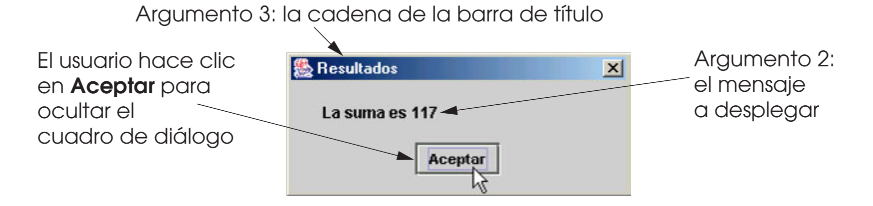
de la instrucción anterior utiliza el operador **+** para "sumar" una cadena (la literal **"La suma es ")** y **suma** (la variable **int** que contiene el resultado de la suma en la línea 28). Java tiene una versión del operador + para *concatenación de cadenas* que permite concatenar una cadena y un valor de otro tipo de dato (incluso otra cadena), el resultado de esta operación es una nueva cadena (por lo general más grande). Si asumimos que la suma contiene el valor **117**, la expresión se evalúa de la siguiente manera: Java determina que los dos operandos del operador + (la cadena **"La suma es "** y el entero **suma**) son de tipos diferentes y uno de ellos es una cadena. A continuación, **suma** se convierte automáticamente en una cadena y se concatena con **"La suma es"**, lo cual arroja como resultado **"La suma es 117"**. Esta cadena se despliega en el cuadro de diálogo. Observe que la conversión automática de un entero **suma** solamente ocurre debido a que se concatena con la literal de cadena **"La suma es "**. Observe además que el espacio intermedio entre **es** y **117** es parte de la cadena **"La suma es "**.



### Error común de programación 24.10

*Confundir el operador* **+** *utilizado para la concatenación de cadenas con el operador* **+** *utilizado para la suma puede provocar resultados extraños. Por ejemplo, al asumir que la variable entera* **y** *tiene el valor* **5***, la expresión "***y + 2 = " + y + 2** *arroja como resultado la cadena "***y + 2 = 52***", no "***y + 2 = 7***", debido a que el primer valor de* **y** *se concatena con la cadena "***y + 2 =***", después el valor* **2** *se concatena con la cadena más grande "***y + = 5***". La expresión* **"y + 2 =" + (y + 2)** *produce el resultado deseado.*

La versión del método **showMessageDialog** utilizada en la figura 24.7 es diferente de la que explicamos en la figura 24.4, en la que se requieren cuatro argumentos. El siguiente cuadro de diálogo explica dos de los cuatro argumentos. Así como en la primera versión, el primer argumento siempre será **null** hasta que expliquemos la clase **JOptionPane** con detalle en el capítulo 29. El segundo argumento es el mensaje a desplegar. El tercer argumento es la cadena a desplegar en la barra de título del cuadro de diálogo. El cuarto argumento **(JOptionPane.PLAIN_MESSAGE)** es un valor que indica el tipo de cuadro mensaje a desplegar, este tipo de mensaje no despliega un icono a la izquierda del mensaje.




En la figura 24.8 mostramos los tipos de diálogos de mensaje. Todos los tipos de diálogo de mensaje, excepto **PLAIN_MESSAGE**, despliegan un icono para el usuario que indica el tipo de mensaje.

| Tipo de diálogo de mensaje | Icono | Descripción |
|---|---|---|
| **JOptionPane.ERROR_MESSAGE** | | Despliega un diálogo que indica un error en la aplicación del usuario. |
| **JOptionPane.INFORMATION_MESSAGE** | | Despliega un diálogo con un mensaje de información sobre la aplicación para el usuario; el usuario simplemente puede ignorar el diálogo. |
| **JOptionPane.WARNING_MESSAGE** | | Despliega un diálogo que advierte al usuario de la aplicación acerca de un problema potencial. |
| **JOptionPane.QUESTION_MESSAGE** | | Despliega un diálogo que coloca una pregunta para el usuario de la aplicación. Por lo general, requiere una respuesta tal como hacer clic en el botón **Sí** o **No**. |
| **JOptionPane.PLAIN_MESSAGE** | sin icono | Despliega un diálogo que simplemente contiene un mensaje sin icono. |

**Figura 24.8** Constantes de **JOptionPane** para los diálogos de mensaje.

## 24.6 Applets de ejemplo del Java 2 Software Development Kit

Ahora veamos otro tipo de programa en Java: los applets. Comenzaremos nuestra introducción a los applets de Java considerando varios ejemplos proporcionados dentro del Java 2 Software Development Kit (J2SDK) versión 1.4. El applet demuestra una pequeña porción de las poderosas capacidades de Java. Cada programa de ejemplo del J2SDK viene también con el *código fuente* en Java; los archivos **.java** que contienen los applets de Java. Este código fuente será útil mientras progresa en el conocimiento de Java; puede leer el código fuente proporcionado para aprender nuevas y excitantes características de Java. Recuerde, de inicio todos los programadores aprenden nuevos conceptos de programación al imitar el uso de esos conceptos dentro de programas existentes. El J2SDK viene con muchos de esos programas y existe un enorme número de recursos de Java en Internet a través de la World Wide Web que incluyen el código fuente en Java.

Los programas de demostración proporcionados con el J2SDK se localizan en el directorio de instalación de J2SDK dentro de un subdirectorio llamado **demo**. Para el Java 2 Software Development Kit versión 1.4, la ubicación predeterminada para el directorio **demo** en Windows es:

```
c:\j2sdk1.4.1\demo
```

En UNIX/Linux/Mac OS X, es el directorio en el que instala el J2SDK seguido por **j2sdk1.4.1/demo**, por ejemplo,

```
/usr/local/j2sdk1.4.1/demo
```

Para otras plataformas, debe haber una estructura de directorios (o carpetas) similar. En este capítulo asumimos que la J2SDK se instala en **c:\j2sdk1.4.1** en Windows y en su directorio inicial (home) **~/j2sdk1.4.1** en UNIX/Linux/MAC OS X.[1]

Si usted utiliza la herramienta de desarrollo de Java que no contiene los demos de Java, puede descargar el J2SDK (con demos) desde el sitio Web de Sun Microsystems

```
java.sun.com/j2se/1.4.1/
```

### 24.6.1 El applet TicTacToe

El primer applet que demostramos a partir de los demos del J2SDK es el applet llamado **TicTacToe**, el cual nos permite jugar Tic-Tac-Toe (Gato) en contra de la computadora. Para ejecutar este applet, abra una ventana de comando (el indicador de MS-DOS en Windows 95/98/ME, el símbolo del sistema en Windows NT/2000/XP, o una ventana de terminal en UNIX/Linux/Mac OS X) y cambie de directorio hacia el directorio correspondiente al demo del J2SDK. Cada sistema operativo que mencionamos aquí utiliza el comando ***cd*** para *cambiar de directorio*. Por ejemplo,

```
cd c:\j2sdk1.4.1\demo
```

cambia el directorio activo hacia **demo** en Windows y

```
cd ~/j2sdk1.4.1/demo
```

cambia el directorio activo hacia **demo** en UNIX/Linux/Mac OS X.

El directorio **demo** contiene varios subdirectorios. Usted puede listar estos directorios al escribir **dir** en la ventana de comandos de Windows, o el comando **ls** en UNIX/Linux/Mac OS X. Explicaremos los directorios ***applets*** y ***jfc***. El directorio **applets** contiene muchos applets de demostración. El directorio **jfc** (Java Foundation Classes) contiene muchos ejemplos de las características de gráficos y GUI de Java (algunos de estos ejemplos también son applets). Cambie el directorio activo al directorio applet mediante el siguiente comando

```
cd applets
```

ya sea en Windows o en UNIX/Linux/Mac OS X.

Liste el contenido del directorio **applets** para ver los nombres de directorio para los applets de demostración. La figura 24.9 proporciona una breve descripción de cada ejemplo.

---

1. Es posible que necesite actualizar estas ubicaciones para reflejar el directorio de instalación que eligió y la unidad de disco, o una versión diferente de J2SDK.

| Ejemplo | Descripción |
|---|---|
| **`Animator`** | Ejecuta una de cuatro animaciones diferentes. |
| **`ArcTest`** | Demuestra el dibujo de arcos. Usted puede interactuar con el applet para modificar los atributos del arco que se despliega. |
| **`BarChart`** | Dibuja un gráfico sencillo de barras. |
| **`Blink`** | Despliega texto intermitente en diferentes colores. |
| **`CardTest`** | Demuestra varios componentes GUI y una variedad de formas en las que pueden organizarse los componentes GUI en la pantalla. (La organización de los componentes GUI también se conoce como *diseño* de los componentes GUI.) |
| **`Clock`** | Dibuja un reloj de manecillas "rotantes", la fecha y la hora actuales. El reloj se actualiza una vez por segundo. |
| **`DitherTest`** | Demuestra el dibujo con la técnica de gráficos conocida como *tramado*, la cual permite la transformación gradual de un color a otro. |
| **`DrawTest`** | Permite al usuario arrastrar el ratón para dibujar líneas y puntos de diferentes colores en el applet. |
| **`Fractal`** | Dibuja un fractal. Por lo general, los fractales requieren cálculos complejos para determinar la manera en que se despliegan. |
| **`GraphicsTest`** | Dibuja una variedad de figuras para ilustrar las capacidades gráficas de Java. |
| **`GraphLayout`** | Dibuja un grafo que consta de muchos nodos (representados como rectángulos) conectados mediante líneas. Arrastre un nodo para que vea cómo los demás nodos se ajustan en la pantalla y para demostrar sus complejas interacciones gráficas. |
| **`ImageMap`** | Demuestra una imagen con *puntos activos*. Al colocar el apuntador del ratón sobre ciertas áreas de la imagen resalta el área y se despliega un mensaje en la esquina inferior derecha de la ventana del **`appletviewer`**. Coloque el apuntador del ratón sobre la boca de la imagen para escuchar al applet decir "hi" (hola). |
| **`JumpingBox`** | Mueve un rectángulo de manera aleatoria alrededor de la pantalla. ¡Intente atraparlo haciendo clic sobre él con el ratón! |
| **`MoleculeViewer`** | Presenta una vista tridimensional de varias moléculas químicas diferentes. Arrastre el ratón para ver la molécula desde diferentes ángulos. |
| **`NervousText`** | Dibuja texto que salta alrededor de la pantalla. |
| **`SimpleGraph`** | Dibuja una curva compleja. |
| **`SortDemo`** | Compara tres métodos de ordenamiento. Clasifica la información en orden; como palabras en orden alfabético. Cuando usted ejecuta el applet, aparecen tres ventanas del **`appletviewer`**. Haga clic en cada una para comenzar el ordenamiento. Observe que los ordenamientos operan a velocidades diferentes. |
| **`SpreadSheet`** | Muestra una hoja de cálculo sencilla con líneas y columnas. |
| **`SymbolTest`** | Dibuja un carácter desde el conjunto de caracteres de Java. |
| **`TicTacToe`** | Permite al usuario jugar Gato en contra de la computadora. |
| **`WireFrame`** | Dibuja una figura en tres dimensiones como una malla alámbrica. Arrastre el ratón para ver la figura desde diferentes ángulos. |

**Figura 24.9** Ejemplos del directorio **`applets`**.

Cambie los directorios al subdirectorio **`TicTacToe`**. En este directorio existe un archivo HTML (**`example1.html`**) que se utiliza para ejecutar el applet. En la ventana de comando, escriba

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. Esto ejecuta el **`appletviewer`**. El **`appletviewer`** carga el archivo HTML especificado como un *argumento* de *la línea de comando* (**`example1.html`**), determina desde el archivo cuál applet

cargar (explicaremos los detalles de los archivos HTML en la sección 24.7) y comienza la ejecución del applet. La figura 24.9 muestra varias capturas de pantalla del juego del Gato con este applet.

**Tip para prevenir errores 24.3**

*Si el comando* **`appletviewer`** *no funciona y/o el sistema indica que el comando* **`appletviewer`** *no se encuentra, la variable de ambiente* **`PATH`** *podría no estar definida apropiadamente en su computadora. Revise las direcciones de instalación para el Java 2 Software Development Kit para asegurarse de que la variable de ambiente está correctamente definida para su sistema (en algunas computadoras, podría ser necesario reiniciar el equipo después de definir la variable de ambiente* **`PATH`***).*

Usted es el jugador **X**. Para interactuar con el applet, apunte el ratón sobre el cuadro en donde desea colocar la **X** y haga clic con el botón del ratón (por lo general, con el botón izquierdo). El applet reproduce un sonido (suponemos que su computadora soporta la reproducción de audio) y coloca una **X** en el cuadrado si éste está vacío. Si el cuadrado está ocupado, éste es un movimiento inválido y el applet ejecuta un sonido diferente que indica que usted no puede realizar ese movimiento específico. Después de hacer un movimiento válido, el applet responde haciendo su propio movimiento (esto sucede de inmediato).

Para jugar de nuevo, ejecute de nuevo el applet haciendo clic en el *menú* **Subprograma** del **`appletviewer`**, y seleccionar el elemento de menú **Volver a cargar**. Para finalizar el **`appletviewer`**, haga clic en el menú **Subprograma** y seleccione el *elemento de menú* **Salir**.

## 24.6.2 El applet `DrawTest`

El siguiente applet que explicaremos le permitirá dibujar líneas y puntos de diferentes colores. Para dibujar, simplemente arrastre el ratón sobre el applet y mantenga oprimido el botón mientras arrastra el ratón. Para este ejemplo, cambie al directorio **`applets`**, y después al subdirectorio **`DrawTest`**. En dicho directorio se encuentra el archivo **`example1.html`** que se utiliza para ejecutar el **`applet`**. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

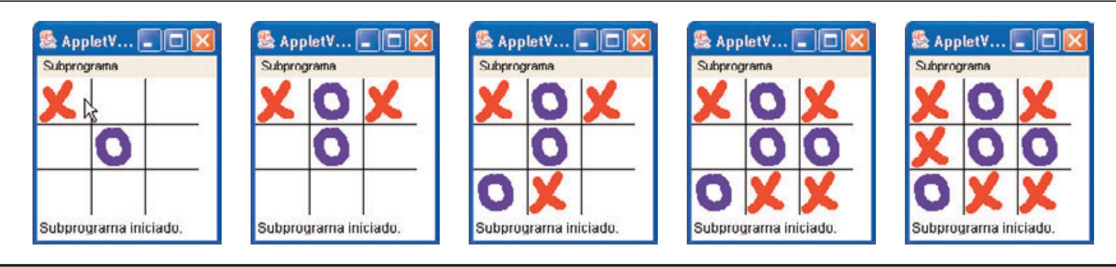



**Figura 24.10** Ejecución de ejemplo del applet **`TicTacToe`**.

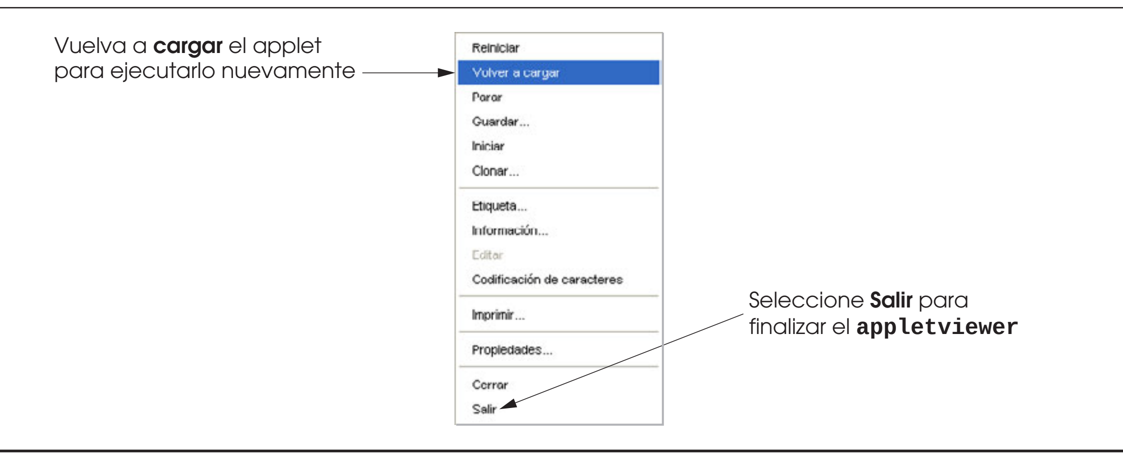



**Figura 24.11** Selección de **`Volver a cargar`** del menú **`Subprograma`** del **`appletviewer`**.

y oprima la tecla *Entrar*. Esto ejecuta el **appletviewer**. El **appletviewer** carga el archivo HTML especificado como su argumento en la línea de comando (de nuevo, **example1.html**), determina en el archivo cuál applet cargar y comienza la ejecución del applet. En la figura 24.12 puede apreciar una captura de pantalla de este applet después de dibujar algunas líneas y puntos.

La figura predeterminada a dibujar es una línea y el color predeterminado es el negro, de modo que usted puede dibujar líneas negras al instante con solo arrastrar el ratón a lo largo del applet. Para arrastrar el ratón, presione el botón del ratón, manténgalo presionado y muévalo. Observe que la línea sigue al apuntador del ratón alrededor del applet. La línea no se vuelve permanente hasta que suelta el botón del ratón. Puede comenzar una nueva línea, repitiendo el proceso.

Seleccione un color haciendo clic en el círculo interno de uno de los rectángulos coloreados que se encuentran en la parte inferior del applet. Puede seleccionar entre el rojo, el verde, el azul, el anaranjado y el negro. Por lo general, a los componentes GUI utilizados para presentar estas opciones se les conoce como *botones de opción*. Si se imagina el estéreo de un automóvil, solamente se puede seleccionar una estación de radio a la vez. De manera similar, solamente se puede dibujar en un color a la vez.

Intente modificar la figura de **Líneas** a **Puntos**, haciendo clic en la flecha que apunta hacia abajo que aparece a la derecha de la palabra **Lines** en la parte inferior del applet. La lista desplegable del componente GUI contiene dos opciones, **Lines** y **Points**. Para seleccionar **Points**, haga clic en la palabra **Points** de la lista. El componente GUI cierra la lista, y ahora **Points** será la figura actual. Por lo general, a este componente GUI se le conoce cómo *opción*, *cuadro combinado* o *lista desplegable*.

Para comenzar un nuevo dibujo, seleccione **Volver a cargar** desde el menú **Subprograma** del **Appletviewer**. Para finalizar el applet, seleccione **Salir** del menú **Subprograma** del **appletviewer**.

## 24.6.3 El applet Java2D

El último applet que explicamos (figura 24.13), antes de definir nuestros propios applets, muestra muchas de las nuevas y complejas capacidades de dos dimensiones incluidas en Java 2; conocida como el *API Java2D*. Para este ejemplo, cambie al directorio **jfc** que se encuentra en el directorio **demo** del J2SDK, después cambie al directorio **Java2D** (usted puede moverse en el árbol de directorios hacia **demo** con el comando "**cd ..**", tanto

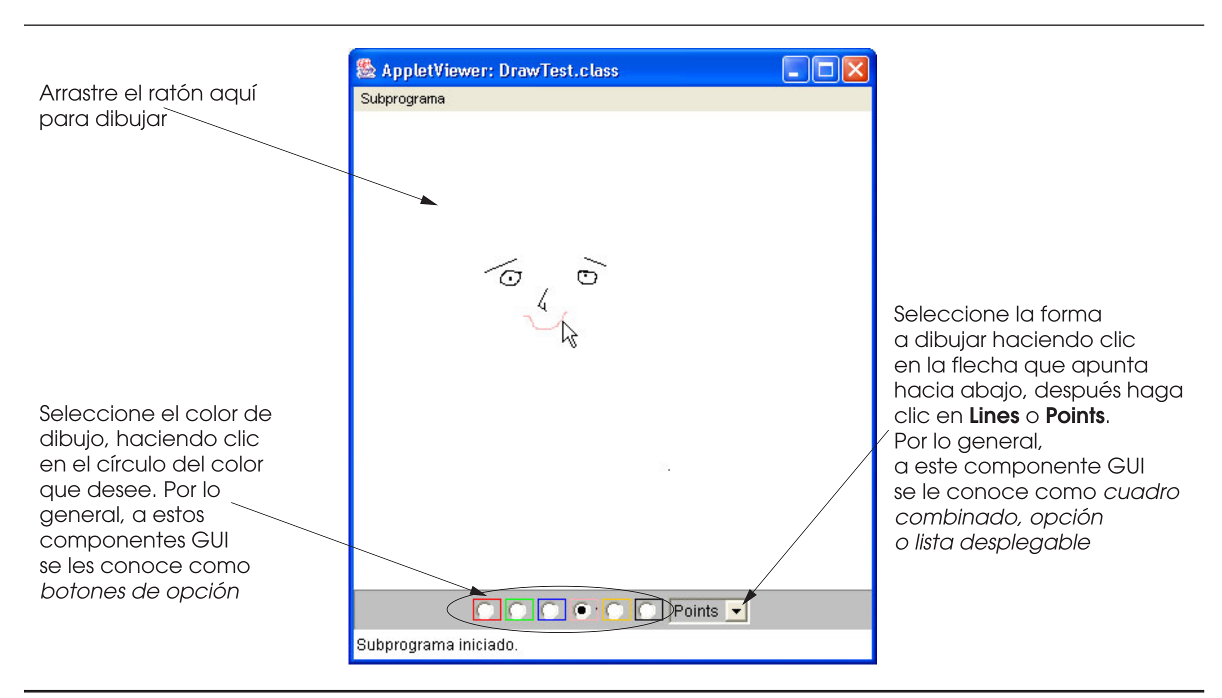



**Figura 24.12** Ejemplo de la ejecución del applet **DrawTest**.

en Windows como en UNIX). En dicho directorio se encuentra un archivo HTML (**`Java2Demo-Applet.html`**), que se utiliza para ejecutar el applet. En la ventana de comando escriba

```
appletviewer Java2DemoApplet.html
```

y oprima la tecla *Entrar*. Esto ejecuta el **`appletviewer`**. El **`appletviewer`** carga el archivo HTML especificado como su argumento de línea de comando (**`Java2DemoApplet.html`**), determina desde el archivo cuál applet cargar y comienza la ejecución del applet. Este **`demo`** en particular se lleva cierto tiempo en cargar, debido a que es bastante grande. La figura 24.12 muestra una captura de pantalla de una de las muchas demostraciones de este applet con respecto a las nuevas capacidades para gráficos de dos dimensiones de Java.

En la parte superior de este demo se aprecian fichas que parecen carpetas de un archivero. Este demo proporciona 11 fichas diferentes con muchas características diferentes en cada ficha. Para cambiar a una parte diferente del demo, simplemente haga clic en una de las fichas. También intente modificar las opciones en la esquina superior derecha del applet. Algunas de éstas afectan la velocidad a la cual el applet dibuja los gráficos. Por ejemplo, haga clic en el cuadro pequeño con una marca en él (un componente GUI conocido como *cuadro de verificación*) a la izquierda de la palabra **Anti-Aliasing** para deshabilitar la técnica de distorsión de gráficos (una técnica gráfica para producir gráficos en pantalla más suaves, en los que los límites de las figura se hacen más difusos). Cuando se deshabilita esta característica (es decir, el *cuadro de verificación* se desmarca), se incrementa la velocidad de animación de las figuras animadas en el fondo del demo que aparece en la figura 24.13. Esto se debe a que una figura animada con distorsión toma más tiempo para dibujarse que una figura animada sin distorsión.

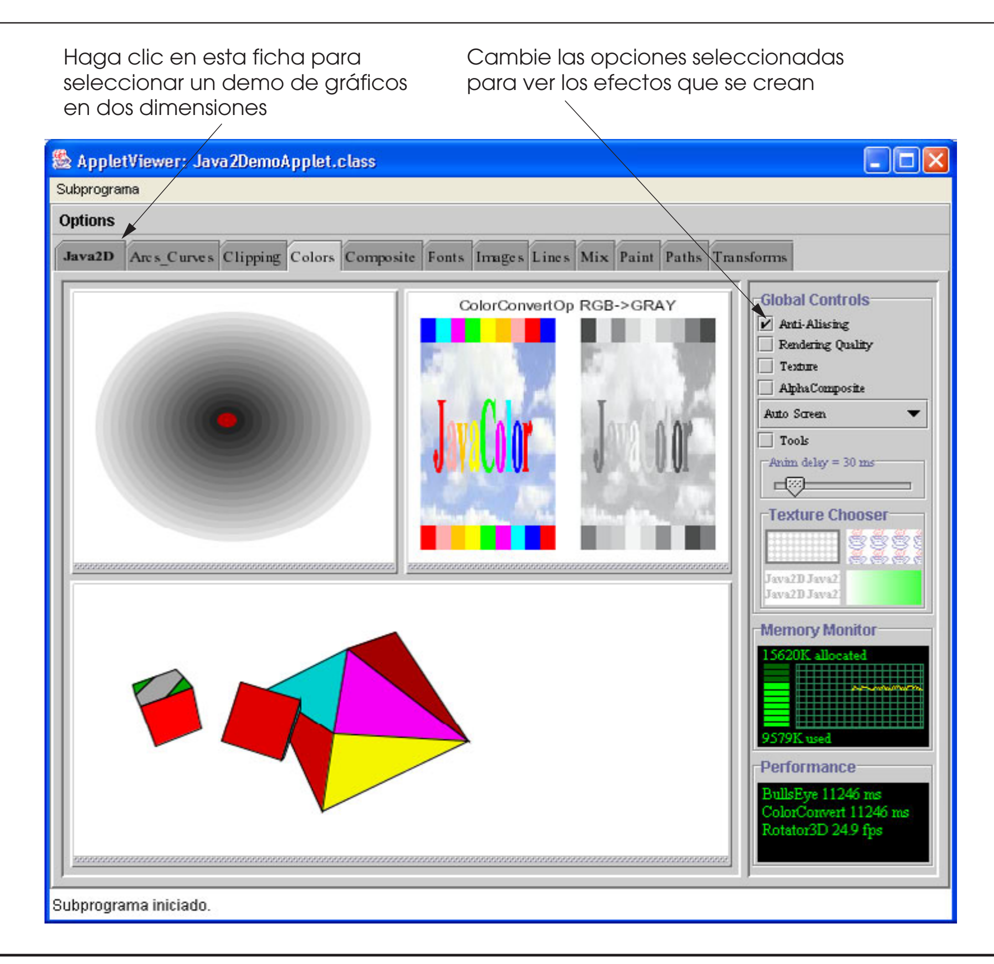
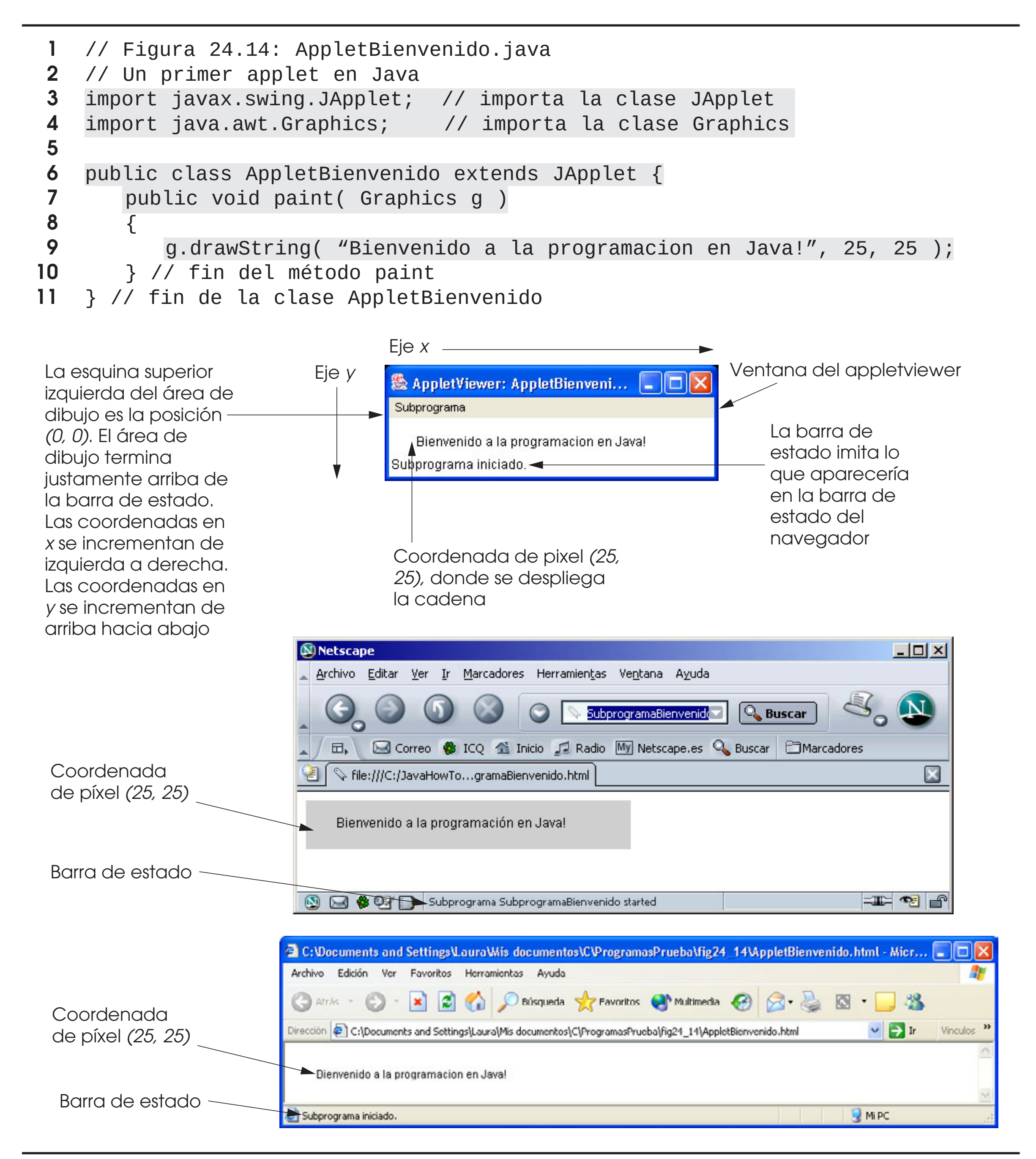



**Figura 24.13** Ejecución de ejemplo del applet **`Java2D`**.

## 24.7 Un applet sencillo en Java: Cómo dibujar una cadena

Ahora, comenzamos con algunos applets propios. Recuerde que sólo estamos comenzando; tenemos que aprender muchas cosas más antes de que podamos escribir applets similares a las que mostramos en las secciones 24.6. Sin embargo, en capítulos posteriores abordaremos muchas de las mismas técnicas.

Comencemos considerando un applet sencillo que imita el programa de la figura 24.2 al desplegar la cadena **`“Bienvenido a la programacion en Java!”`**. El applet y su salida en la pantalla aparecen en la figura 24.14. La figura 24.15 muestra y explica el documento HTML para cargar el applet en el **`appletviewer`**.



```
// Figura 24.14: AppletBienvenido.java
// Un primer applet en Java
import javax.swing.JApplet;  // importa la clase JApplet
import java.awt.Graphics;    // importa la clase Graphics

public class AppletBienvenido extends JApplet {
   public void paint( Graphics g )
   {
      g.drawString( “Bienvenido a la programacion en Java!”, 25, 25 );
   } // fin del método paint
} // fin de la clase AppletBienvenido
```



**Figura 24.14** Un primer applet en Java, y su salida en pantalla.

```
<html>
<applet code="AppletBienvenido.class" width=300 height=30>
</applet>
</html>
```

**Figura 24.15** Archivo **AppletBienvenido.html**, el cual carga la clase **AppletBienvenido** de la figura 24.14 dentro del **appletviewer**.

Este programa muestra muchas características importantes de Java. Consideraremos con detalle cada línea del programa. La línea 9 hace el "trabajo real", a saber: el dibujo de la cadena **Bienvenido a la programacion en Java!** en la pantalla. Sin embargo, consideremos cada línea del programa en orden. Las líneas 1 y 2

```
// Figura 24.14: AppletBienvenido.java
// Un primer applet en Java
```

comienzan con //, lo que indica que el resto de cada línea es un comentario. El comentario en la línea 1 indica el número de la figura y el nombre del código fuente del applet. El comentario **Un primer applet en Java** de la línea 2 simplemente describe el propósito del programa.

Como dijimos anteriormente, Java contiene muchas piezas predefinidas llamadas clases (o tipos de datos) que están agrupadas dentro de paquetes en el API de Java. Las líneas 3 y 4

```
import javax.swing.JApplet;       // importa la clase JApplet
import java.awt.Graphics;         // importa la clase Graphics
```

son instrucciones para importar que le indican al compilador en dónde localizar las clases requeridas para compilar este applet de Java. Estas líneas específicas le indican al compilador que la clase ***JApplet*** se encuentra ubicada en el paquete **javax.swing** y que la clase ***Graphics*** se encuentra ubicada en ***java.awt***. Cuando usted crea un applet de Java, por lo general importa la clase **JApplet**. Importamos la clase **Graphics** para que el programa pueda dibujar gráficos (tales como líneas, rectángulos, elipses y cadenas de caracteres) en un applet de Java (o en una aplicación, más adelante en el libro). [*Nota:* Existe una clase más antigua llamada ***Applet*** del paquete ***java.applet*** que no se utiliza con los componentes GUI más novedosos del paquete **javax.swing**. En el libro, solamente utilizaremos la clase **JApplet** para los applets.]

Cada pieza del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en el API de Java se almacenan en el directorio **java** o **javax** que contiene muchos subdirectorios, incluso **awt** y **swing**. [*Nota:* Si busca estos directorios en el disco, no los encontrará debido a que están almacenados dentro de un archivo comprimido especial llamado *Java archive file (JAR)*. Dentro de la estructura de directorios de instalación de J2SDK es un archivo llamado ***rt.jar*** que contiene los archivos **.class** para el API completo de Java.]

Así como las aplicaciones, cada applet de Java está compuesto por al menos una definición de clase. Una característica clave de las definiciones de clases que no mencionamos antes es que usted rara vez creará una definición de clase "desde cero". De hecho, cuando crea una definición de una clase, por lo general utiliza piezas de una definición de clase existente. Java utiliza la *herencia* (que explicaremos con detalle en el capítulo 20) para crear nuevas clases a partir de definiciones de clases existentes. La línea 6

```
public class AppletBienvenido extends JApplet {
```

inicia la definición de la clase **AppletBienvenido**. Una vez más, la palabra reservada **class** introduce la definición de una clase e inmediatamente es seguido por el nombre de la clase (**AppletBienvenido**, en este caso). La *palabra reservada* **extends** seguida por el nombre de la clase indica la clase (en este caso **JApplet**), a partir de la cual nuestra nueva clase hereda las piezas existentes. En esta relación de herencia, a **JApplet** se le conoce como la *superclase* o la *clase base*, y a **AppletBienvenido** se le llama la *subclase* o la *clase derivada*. En el capítulo 27 explicaremos con detalle la herencia. Utilizar la herencia en este punto da como resultado una nueva definición de clase que tiene los *atributos* (datos) y *comportamientos* (métodos) de la clase **JApplet**, así como las nuevas características que agregamos en nuestra definición de la clase **AppletBienvenido** (específicamente, la habilidad de desplegar en la pantalla **Bienvenido a la programacion en Java!**).

Un beneficio importante de extender la clase **`JApplet`** es que cualquiera tiene definido “lo que significa un applet”. El **`appletviewer`** y los navegadores de la World Wide Web que soportan applets esperan que todos los applets de Java tengan ciertas capacidades (atributos y comportamientos), y la clase **`JApplet`** proporciona todas esas capacidades; los programadores no necesitan definir todas las capacidades por su cuenta (de nuevo, los programadores no necesitan “reinventar la rueda”). De hecho, un applet requiere alrededor de 200 métodos diferentes para completar su definición. Hasta este punto, en nuestros programas hemos definido un método para cada programa. Si hubiéramos tenido que definir cerca de 200 métodos sólo para desplegar **`Bienvenidos a la programacion en Java!`**, ¡probablemente nunca hubiéramos definido un applet! Con tan sólo utilizar **`extends`** para heredar de la clase **`JApplet`**, ahora todos los métodos de **`JApplet`** son parte de nuestra clase **`AppletBienvenido`**.

El mecanismo de herencia es fácil de utilizar: el programador no necesita conocer cada detalle de la clase **`JApplet`** o cualquier otra clase a partir de la que hereda. El programador sólo necesita conocer que la clase **`JApplet`** ya tiene definidas las capacidades requeridas para crear el applet mínimo. Sin embargo, para hacer mejor uso de cualquier clase, el programador debe estudiar todas las capacidades de la clase que se extiende.



### Buena práctica de programación 24.15

*Investigue cuidadosamente las capacidades de cualquier clase en la documentación del API de Java, antes de heredar a una subclase. Esto ayuda a asegurar que el programador no redefine por descuido una capacidad que ya está proporcionada.*

Las clases se utilizan como “plantillas” o “anteproyectos” para *instanciar* (o *crear*) *objetos* que se utilizarán en el programa. Un objeto (o *instancia*) reside en la memoria de la computadora y contiene información que utiliza el programa. Por lo general, el término objeto implica que los atributos (datos) y los comportamientos (métodos) están asociados con el objeto. Los métodos del objeto utilizan atributos para proporcionar servicios útiles para el *cliente del objeto* (es decir, el código que llama a los métodos).

Nuestra clase **`AppletBienvenido`** se utiliza para crear un objeto que implementa los atributos y comportamientos del applet. El comportamiento predeterminado del método **`paint`** de la clase **`JApplet`** no hace cosa alguna. La clase **`AppletBienvenida`** *redefine* (o *reemplaza*) el comportamiento **`paint`** que dibuja un mensaje en la pantalla. Cuando un **`appletviewer`** o un navegador le indica al applet que se “dibuje a sí mismo” en la pantalla mediante la llamada al método **`paint`**, nuestro mensaje **`Bienvenido a la programacion en Java!`** aparece, en vez de una pantalla en blanco.

El **`appletviewer`** o navegador en el que se ejecuta el applet es responsable de la creación del objeto (instancia) de la clase **`AppletBienvenido`**. [*Nota:* Con frecuencia los términos instancia y objeto se utilizan indistintamente.] La palabra reservada ***`public`*** de la línea 6 es necesaria para permitir al navegador la creación de un objeto de la clase **`AppletBienvenido`** y la ejecución del applet. La clase que hereda de **`JApplet`** para crear un applet debe ser una clase pública. En el capítulo 26, explicamos con detalle la palabra reservada **`public`** y las palabras reservadas relacionadas (tales como **`private`** y **`protected`**). Por ahora, simplemente le pediremos que comience todas las definiciones de las clases con la palabra reservada **`public`**, hasta que la expliquemos en el capítulo 26.

Cuando guarda la clase **`public`** en un archivo, el nombre de la clase se utiliza como parte del nombre del archivo. Para nuestro applet, el nombre del archivo debe ser **`AppletBienvenido.java`**. Observe que el nombre del archivo debe deletrearse exactamente como en el nombre de la clase y debe tener la extensión de nombre de archivo **`.java`**.



### Tip para prevenir errores 24.4

*El mensaje de error del compilador “Public class ClassName must be defined in a file called* ClassName*.java” indica que 1) el nombre del archivo no coincide exactamente con el nombre de la clase* ***`public`*** *en el archivo (incluidas todas las letras mayúsculas y minúsculas), o 2) que usted escribió el nombre de la clase de manera incorrecta cuando compiló la clase (el nombre debe deletrearse con las letras mayúsculas y minúsculas apropiadas).*

Al final de la línea 6, la llave izquierda, **`{`**, comienza el cuerpo de la definición de la clase. La llave derecha correspondiente, **`}`**, de la línea 11 termina la definición de la clase. La línea 7

```
public void paint( Graphics g )
```

comienza la definición del *método* **paint** del applet. El método **paint** es uno de los tres métodos (comportamientos) que con seguridad serán llamados cuando comience la ejecución del applet. Estos tres métodos son **init** (que explicaremos más adelante en este capítulo), **start** (que explicaremos más adelante en el libro) y **paint**, y seguramente serán llamados en ese orden. Estos métodos se llaman desde el **appletviewer** o desde el navegador en el que se ejecuta el applet. Su clase applet obtiene una versión "libre" de cada uno de estos métodos desde la clase **JApplet,** cuando usted especifica **extends JApplet** en la primera línea de su definición de la clase applet. Existen muchos otros métodos que seguramente se llamarán durante la ejecución del applet, explicaremos dichos métodos en el capítulo 25.

La versión libre de cada uno de estos tres métodos se define con un cuerpo vacío (es decir, de manera predeterminada, estos métodos no realizan tarea alguna). Una de las razones por las que heredamos todos los applets de la clase **JApplet** es para obtener nuestras copias libres de los métodos que se llaman de manera automática durante la ejecución de un applet (y también muchos otros métodos).

¿Por qué desearía una copia gratis de un método que no hace cosa alguna? La secuencia de inicio predefinida para las llamadas a los métodos hechas por el **appletviewer** o por el navegador para cada applet siempre es **init**, **start** y **paint**; esto proporciona a un programador de applets una secuencia de inicio garantizada para las llamadas a los métodos al comenzar la ejecución de cada applet. No todos los applets necesitan estos tres métodos. Sin embargo, el **appletviewer** o el navegador esperan que cada uno de estos métodos esté definido, de modo que pueda proporcionar una secuencia de inicio consistente para un applet. [*Nota:* Esto es similar para las aplicaciones que siempre inician la ejecución en **main**.] Heredar las versiones predeterminadas para estos métodos garantiza que el navegador pueda tratar a cada objeto del applet de manera uniforme al llamar a **init**, **start**, y **paint** cuando comience la ejecución de los applets. Además, el programador puede concentrarse sólo en la definición de los métodos requeridos para un applet en especial.

Las líneas 7 a 10 son definiciones de **paint**. La tarea o método **paint** sirve para dibujar gráficos (tal como líneas, elipses y cadenas de caracteres) en la pantalla. La palabra reservada **void** indica que este método no devuelve resultado alguno cuando termina su tarea. El conjunto de paréntesis después de **paint** define la *lista de parámetros* del método. Recuerde que la lista de parámetros es en donde los métodos reciben los datos necesarios para llevar a cabo su tarea. Por lo general, los datos pasan del programador al método a través de la *llamada al método* (también conocida como *invocación de un método* o *envío de un mensaje*). Por ejemplo, en la figura 24.4, pasamos los datos a **JOptionPane.showMessageDialog**, incluso el mensaje a desplegar y el tipo de diálogo de mensaje. Sin embargo, el método **paint**, el cual es llamado para que podamos dibujar en el área de visualización del applet en la pantalla, recibe automáticamente la información necesaria cuando se llama al método. La lista de parámetros del método **paint** indica que requiere un objeto **Graphics** (llamado **g**) para realizar su tarea. El objeto **Graphics** se utiliza en **paint** para dibujar gráficos sobre el applet. La palabra reservada **public** al principio de la línea 7 es necesaria para que el **appletviewer** o el navegador puedan llamar a su método **paint**. Por ahora, todas las definiciones de métodos deben comenzar con la palabra reservada **public**. En el capítulo 26 presentaremos otras alternativas.

La llave izquierda, **{**, de la línea 8 comienza la definición del cuerpo del método. La llave derecha correspondiente, **}**, de la línea 10 termina la definición del cuerpo del método. La línea 9

```
g.drawString( "Bienvenido a la programacion en Java!", 25, 25 );
```

es una instrucción que indica a la computadora que realice una acción (o tarea), a saber, para desplegar los caracteres de la cadena de caracteres **Bienvenido a la programacion en Java!** en el applet. Esta instrucción utiliza el método **drawString** definido por la clase **Graphics** (esta clase define todas las capacidades para dibujar gráficos de un programa en Java, como el dibujo de cadenas de caracteres y el dibujo de figuras tales como rectángulos, elipses y líneas). El método **drawString** se llama mediante el uso del objeto **g** de **Graphics** (en la lista de parámetros de **paint**) seguido por el operador punto (**.**), seguido por el nombre del método **drawString**. El nombre del método es seguido por un conjunto de paréntesis que contienen la lista de argumentos que **drawString** necesita para realizar su tarea.

El primer argumento para **drawString** es el **String** a dibujar. Los dos últimos argumentos en la lista, **25** y **25**, son las *coordenadas* (o la *posición*) en la que debe dibujarse la esquina inferior izquierda de la cadena en el área de pantalla del applet. Las coordenadas se miden en *pixeles* a partir de la esquina superior izquierda del applet (la esquina superior izquierda del área blanca correspondiente a la pantalla de captura de la figura 24.14).

Un píxel ("elemento de dibujo") es la unidad de despliegue para la pantalla de su computadora. En una pantalla a color, un píxel aparece como un punto de color en la pantalla. Muchas computadoras personales tienen 640 pixeles para el ancho de la pantalla y 480 pixeles de alto, lo que da un total de 640 por 480 o 307,200 pixeles desplegables. Muchas pantallas de computadora tienen mejores resoluciones de pantalla, es decir, tiene más pixeles para el ancho y la altura de la pantalla. Mientras más alta sea la resolución de la pantalla, más pequeño parece el applet en la pantalla. Los métodos de dibujo de la clase **`Graphics`** requieren coordenadas para especificar en dónde dibujar sobre el applet (más adelante en el libro mostraremos el dibujo en las aplicaciones). La primera coordenada es la *coordenada x* (el número de pixeles desde el lado izquierdo del applet), y la segunda coordenada es la *coordenada y* (que representa el número de pixeles desde la parte superior del applet).

Cuando se ejecuta la instrucción anterior, ésta dibuja el mensaje **`Bienvenido a la programacion en Java!`** en el applet en las coordenadas **`25`** y **`25`**. Observe que las comillas que encierran a la cadena de caracteres *no* se despliegan en la pantalla.

Después de definir la clase **`AppletBienvenido`** y de guardar el archivo **`AppletBienvenido.java`**, la clase debe compilarse con el compilador de Java **`javac`**. En la ventana de comandos, escriba el comando

```
javac AppletBienvenido.java
```

para compilar la clase **`AppletBienvenido`**. Si no existen errores de sintaxis, los bytecodes resultantes se almacenan en el archivo **`AppletBienvenido.class`**.

Después de compilar el programa de la figura 24.14, debemos crear un archivo *HTML* (Lenguaje de Marcación de Hipertexto) para cargar el applet dentro del **`appletviewer`** (o del navegador) para ejecutarlo. Por lo general, un archivo HTML termina con la extensión de archivo ***`.html`*** o ***`.htm`***. Los navegadores despliegan el contenido de los documentos que contienen texto (también conocidos como *archivos de texto*). Para ejecutar un applet de Java, debe proporcionar un archivo de texto HTML que indique cuál applet debe cargar el **`appletviewer`** (o el navegador) para su ejecución. La figura 24.15 contiene un archivo HTML sencillo, **`AppletBienvenido.html`**, que se utiliza para cargar el applet dentro del **`appletviewer`** (o del navegador) definido en la figura 24.14. [*Nota:* En este libro, siempre mostramos los applets con el **`appletviewer`**.]



### Buena práctica de programación 24.16

*Siempre pruebe el applet en el* ***`appletviewer`*** *y asegúrese de que se ejecuta correctamente, antes de cargar el applet en un navegador de la World Wide Web. Con frecuencia, los navegadores guardan una copia de un applet en la memoria hasta que termina la sesión actual de navegación (es decir, hasta que se cierran todas las ventanas del navegador). Por lo tanto, si usted modifica un applet, recompílelo, y luego recargue el applet en el navegador; es probable que no vea los cambios, debido a que el navegador aún está ejecutando la versión original del applet. Cierre todas las ventanas de su navegador para eliminar de la memoria la versión anterior del applet. Abra una nueva ventana del navegador y cargue el applet para ver los cambios.*



### Observación de ingeniería de software 24.2

*Si su navegador Web no soporta Java 2, la mayoría de los applets de este libro no se ejecutarán en su navegador. Esto se debe a que la mayoría de los applets de este libro utilizan las características que son nuevas en Java 2, o a que no se proporcionan con los navegadores que soportan Java 1.1.*

Muchos códigos en HTML (o *etiquetas*) vienen en pares. Por ejemplo, las líneas 1 y 4 de la figura 24.15 indican el principio y el final, respectivamente, de las etiquetas HTML en el archivo. Todas las etiquetas HTML comienzan con la *llave angular izquierda*, < y terminan con una *llave angular derecha*, >. Las líneas 2 y 3 son etiquetas especiales en HTML para los applets de Java. Éstas le indican al **`appletviewer`** (o al navegador) que cargue un applet específico y que defina el tamaño del área de despliegue del applet (su *ancho* y su *altura* en pixeles) en el **`appletviewer`** (o el navegador). Por lo general, el applet y su archivo HTML correspondiente se almacenan en el mismo directorio en el disco.

Por lo general, un archivo HTML se carga en el navegador desde una computadora conectada a Internet diferente a la suya. Sin embargo, los archivos HTML también pueden residir dentro de su computadora (como lo demostraremos en la sección 24.6). Siempre que se carga un archivo HTML que especifica la ejecución de un applet dentro del **`appletviewer`** (o del navegador), el **`appletviewer`** (o el navegador) carga el archi-

vo (o archivos) **.class** del applet desde el mismo directorio en la computadora desde la que se cargó el archivo HTML.

La *etiqueta* **<applet>** tiene diversos componentes. El primer componente de la *etiqueta* **<applet>** de la línea 2 (**código="AppletBienvenido.class"**) indica que el archivo **AppletBienvenido.class** contiene la clase compilada del applet. Recuerde, cuando compila sus programas en Java, cada clase se compila en un archivo aparte que tiene el mismo nombre que la clase, y termina con la extensión **.class**. El segundo y el tercer componente de la *etiqueta* **<applet>** indican el **ancho (width)** y la **altura (height)** del applet en pixeles. La esquina superior izquierda del área de visualización siempre es la coordenada 0 en *x*, y la coordenada 0 en *y*. El ancho de este applet es de 300 pixeles. Usted podría querer (o necesitar) utilizar valores más grandes para el ancho y la altura para definir un área de dibujo más grande para sus applets. En la línea 3, la etiqueta **</applet>** finaliza la etiqueta **<applet>** que comenzó en la línea 2. En la línea 4, la etiqueta **</html>** especifica el final de las etiquetas HTML que comienzan en la línea 1 con **<html>**.



### Observación de ingeniería de software 24.3

*Por lo general, cada applet debe tener un tamaño menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones de ancho y altura mínimas).*



### Error común de programación 24.11

*Colocar caracteres adicionales tales como comas (**,**) entre los componentes de la etiqueta <**applet**> puede provocar que el **appletviewer** o el navegador produzcan un mensaje de error que indica un **MissingResourceException** al cargar el applet.*



### Error común de programación 24.12

*Olvidar la etiqueta </**applet**> provoca la carga incorrecta del applet dentro del **appletviewer** o el navegador.*



### Tip para prevenir errores 24.5

*Si usted recibe un mensaje de error **MissingResourceException** durante la carga de un applet dentro del **appletviewer** o del navegador, verifique cuidadosamente la etiqueta <**applet**> en el archivo HTML para ver si hay de errores de sintaxis. Compare su archivo HTML con el archivo de la figura 24.15 para confirmar una sintaxis adecuada.*

El **appletviewer** solamente comprende las etiquetas <**applet**> y </**applet**> de HTML, de modo que en ocasiones se le conoce como el "navegador mínimo" (ignora todas las demás etiquetas de HTML). El **appletviewer** es el lugar ideal para probar la ejecución de un applet y garantizar que dicho applet se ejecuta apropiadamente. Una vez que verifica la ejecución del applet, usted puede agregar las etiquetas <**applet**> y </**applet**> al archivo HTML que será visto por la gente que navega en Internet. Para ejecutar el **AppletBienvenido**, el **appletviewer** se invoca desde la ventana de comandos de la siguiente manera:

```
appletviewer AppletBienvenido.html
```

Observe que el **appletviewer** *requiere* un archivo HTML para cargar un applet. Esto difiere del intérprete **java** para aplicaciones que requieren que el nombre de la clase sea el mismo que el de la aplicación. Además, debe emitirse el comando anterior desde el directorio en el que se localiza el archivo HTML y el archivo **.class** del applet.



### Error común de programación 24.13

*Ejecutar el **appletviewer** con un nombre de archivo que no termina con **.html** o **.htm** provoca un error que evita que el **appletviewer** cargue su applet para ejecución.*



### Tip de portabilidad 24.2

*Verifique sus applets en todos los navegadores utilizados por la gente que ve su applet. Esto le ayudará a asegurar que la gente que vea su applet experimente la funcionalidad que usted espera. [Nota: Una meta del plug-in de Java (que explicaremos posteriormente) es proporcionar la ejecución consistente de un applet en diferentes navegadores.]*

## 24.8 Dos ejemplos más de applets: Cómo dibujar cadenas y líneas

Consideremos ahora otro applet. **`Bienvenido a la programacion en Java!`** puede desplegarse de diferentes maneras. Dos instrucciones **`drawString`** en el método **`paint`** puede imprimir varias líneas como en la figura 24.16 (el archivo HTML correspondiente se encuentra en la figura 24.17).

Observe que cada **`drawString`** puede dibujar en cualquier píxel sobre el applet. La razón por la que las líneas de salida aparecen como muestra la ventana de salida es que especificamos la misma coordenada *x* (**`25`**) para cada **`drawString`**, de modo que las cadenas aparecen alineadas al lado izquierdo, y especificamos coordenadas *y* diferentes (**`25`** en la línea 9 y **`40`** en la línea 10), de modo que las cadenas aparecen en ubicaciones diferentes en el applet. Si invertimos las líneas 9 y 10 en el programa, la salida también aparecerá como se muestra, ya que las coordenadas de los pixeles especificados en cada instrucción **`drawString`** son completamente independientes de las coordenadas especificadas en todas las demás instrucciones **`drawStrings`** (y todas las operaciones de dibujo). El concepto de líneas de texto como mostramos en los métodos **`System.out.println`** y **`JOptionPane.showMessageDialog`** no existe al dibujar gráficos. De hecho, si usted intenta desplegar una cadena que contiene un carácter nueva línea (**`\n`**), solamente verá una pequeña caja negra en la posición de la cadena.

Para hacer más interesante el dibujo, el applet de la figura 24.18 dibuja dos líneas y una cadena. El archivo HTML para cargar el applet dentro del **`appletviewer`** aparece en la figura 24.19.

Las líneas 9 y 10 del método **`paint`**

```
g.drawLine( 15, 10, 210, 10);
g.drawLine( 15, 30, 210, 30);
```

```
1  // Figura 24.16: AppletBienvenido2.java
2  // Cómo desplegar varias cadenas
3  import javax.swing.JApplet;  // importa la clase JApplet
4  import java.awt.Graphics;    // importa la clase Graphics
5
6  public class AppletBienvenido2 extends JApplet {
7     public void paint( Graphics g )
8     {
9        g.drawString( "Bienvenido a", 25, 25 );
10       g.drawString( "la programacion en Java !", 25, 40 );
11    } // fin del método paint
12 } // fin de la clase AppletBienvenido2
```

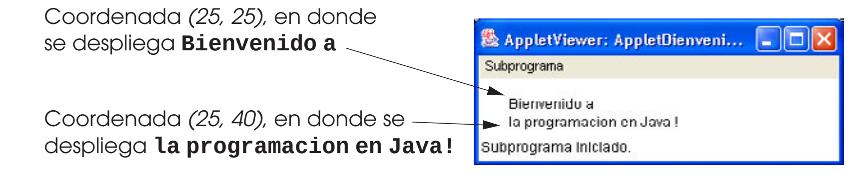



**Figura 24.16** Cómo desplegar varias cadenas.

```
<html>
<applet code="AppletBienvenido2.class" width=300 height=45>
</applet>
</html>
```

**Figura 24.17** Archivo **`AppletBienvenido2.html`**, el cual carga la clase **`AppletBienvenido2`** de la figura 24.16 dentro del **`appletviewer`**.

```
// Figura 24.18: LineasBienvenido.java
// Cómo desplegar texto y líneas
import javax.swing.JApplet;  // importa la clase JApplet
import java.awt.Graphics;    // importa la clase Graphics

public class LineasBienvenido extends JApplet {
   public void paint( Graphics g )
   {
      g.drawLine( 15, 10, 210, 10 );
      g.drawLine( 15, 30, 210, 30 );
      g.drawString( “Bienvenido a la programacion en Java !”, 25, 25 );
   } // fin del método paint
} // fin de la clase LineasBienvenido
```

AppletViewer: LineasBienvenid...
Subprograma
Coordenada *(15, 10)*
Coordenada *(210, 10)*
Bienvenido a la programacion en Java !
Coordenada *(15, 30)*
Coordenada *(210, 30)*
Subprograma iniciado.

**Figura 24.18** Cómo dibujar cadenas y líneas.

```
<html>
<applet code=”LineasBienvenido.class” width=300 height=40>
</applet>
</html>
```

**Figura 24.19** Archivo **`LineasBienvenido.html`**, el cual carga la clase **`LineasBienvenido`** de la figura 24.18 en el **`appletviewer`**.

utilizan el *método* ***`drawLine`*** de la clase **`Graphics`** para indicar que el objeto **`Graphics`** al que hace referencia **`g`** debe dibujar líneas. El método **`drawLine`** requiere cuatro argumentos para representar los dos puntos finales de la línea sobre el applet, la coordenada *x* y la coordenada *y* del primer punto final en la línea, y la coordenada *x* y la coordenada *y* del segundo punto final en la línea. Todos los valores coordinados se especifican con respecto a la coordenada de la esquina superior izquierda *(0, 0)* del applet. Cuando se llama al método **`drawLine`**, éste simplemente dibuja una línea entre dos puntos específcos.

## 24.9 Otro applet de Java: Suma de enteros

Nuestro siguiente applet (figura 24.20) imita la aplicación de la figura 24.7 para sumar dos enteros. Sin embargo, este applet requiere que el usuario introduzca dos *números de punto flotante* (es decir, números con un punto decimal tal como 7.33, 0.0975 y 1000.12345). Para almacenar en memoria números de punto flotante introducimos tipos de datos primitivos ***`double`***, los cuales se utilizan para representar *números de punto flotante* de doble precisión. También existen tipos de datos primitivos ***`float`*** para almacenar *números de punto flotante de precisión sencilla*. Un **`double`** requiere más memoria para almacenar un valor de punto flotante, pero lo almacena con aproximadamente el doble de precisión que un **`float`** (15 dígitos significativos para **`double`** contra siete dígitos significativos para **`float`**).

```
// Figura 24.20: AppletSuma.java
// Suma de dos números de punto flotante
import java.awt.Graphics;     // importa la clase Graphics
```

**Figura 24.20** Un programa de suma “en acción”. (Parte 1 de 2.)

```
import javax.swing.*;          // importa el paquete javax.swing

public class AppletSuma extends JApplet {
   double suma;  // suma de los valores introducidos por el usuario

   public void init()
   {
      String primerNumero,    // primera cadena introducida por el usuario
             segundoNumero;   // segunda cadena introducida por el usuario
      double numero1,         // primer número a sumar
             numero2;         // segundo número a sumar

      // lee el primer número del usuario
      primerNumero =
         JOptionPane.showInputDialog(
            “Introduzca el primer valor de punto flotante” );

      // lee el segundo número del usuario
      segundoNumero =
         JOptionPane.showInputDialog(
            “Introduzca el segundo valor de punto flotante” );

      // convierte los números del tipo String a tipo double
      numero1 = Double.parseDouble( primerNumero );
      numero2 = Double.parseDouble( segundoNumero );

      // suma los números
      suma = numero1 + numero2;
   } // fin del método init

   public void paint( Graphics g )
   {
      // dibuja los resultados con g.drawString
      g.drawRect( 15, 10, 270, 20 );
      g.drawString( “La suma es “ + suma, 25, 25 );
   } // fin del método paint
} // fin de la clase AppletSuma
```

**Figura 24.20** Un programa de suma “en acción”. (Parte 2 de 2.)

```
<html>
<applet code="AppletSuma.class" width=300 height=50>
</applet>
</html>
```

**Figura 24.21** Archivo **`AppletSuma.html`**, el cual carga la clase **`AppletSuma`** de la figura 24.20 dentro del **`appletviewer`**.

Una vez más, utilizamos **`JOptionPane.showInputDialog`** para solicitar información al usuario. El applet después calcula la suma de los valores de entrada y despliega el resultado dibujando una cadena dentro de un rectángulo en un applet. El archivo HTML para cargar este applet dentro del **`appletviewer`** aparece en la figura 24.21.

La línea 3

```
import java.awt.Graphics;   // importa la clase Graphics
```

especifica al compilador en dónde localizar la clase **`Graphics`** (del paquete **`java.awt`**) para utilizarla en esta aplicación. En realidad, la instrucción **`import`** de la línea 3 no es necesaria, si siempre utilizamos el nombre completo de la clase **`Graphics`**, **`java.awt.Graphics`**, el cual incluye el nombre completo del paquete y el nombre de la clase. Por ejemplo, la primera línea del método **`paint`** puede definirse como:

```
public void paint( java.awt.Graphics g )
```



### Observación de ingeniería de software 24.4

*El compilador de Java no necesita instrucciones **`import`** en un archivo de código fuente de Java si el nombre completo de la clase, es decir, el nombre completo del paquete y el nombre de la clase (por ejemplo, **`java.awt.Graphics`**), se especifica cada vez que se utiliza el nombre de la clase en el código fuente.*

La línea 4

```
import javax.swing.*;   // importa el paquete javax.swing
```

especifica al compilador en dónde se ubica el paquete completo **`javax.swing`**. El asterisco (**`*`**) indica que todas las clases en el paquete **`javax.swing`** (tal como **`JApplet`** y **`JOptionPane`**) deben estar disponibles para el compilador, de modo que éste pueda garantizar que utilizamos las clases de manera correcta. Esto permite a los programadores utilizar el *nombre corto* (el nombre de la clase por sí mismo) de cualquier clase del paquete **`javax.swing`** dentro del programa. Recuerde que nuestros dos últimos programas solamente soportan la clase **`JApplet`** del paquete **`javax.swing`**. Importar un paquete completo dentro de un programa también es una notación abreviada para que el programador no tenga que proporcionar una instrucción **`import`** para cada clase del paquete. Recuerde que siempre puede utilizar el nombre completo de cada clase, es decir, **`javax.swing.JApplet`** y **`javax.swing.JOptionPane`**, en lugar de instrucciones **`import`**.



### Observación de ingeniería de software 24.5

*El compilador no carga cada clase en un paquete cuando encuentra una instrucción **`import`** que utiliza la notación **`*`** (por ejemplo, **`javax.swing.*`**) para indicar que se utilizan diversas clases del paquete dentro del programa. El compilador busca en el paquete solamente las clases que utiliza el programa.*



### Observación de ingeniería de software 24.6

*Muchos directorios de paquetes tienen subdirectorios. Por ejemplo, el directorio del paquete **`java.awt`** contiene el subdirectorio **`event`** para el paquete **`java.awt.event`**. Cuando el compilador encuentra una instrucción **`import`** que utiliza la notación **`*`** (por ejemplo, **`java.awt.*`**) para indicar que se utilizan distintas clases del paquete dentro del programa, el compilador no busca el subdirectorio **`event`**. Esto significa que usted no puede definir un **`import`** de **`java.*`** para buscar las clases de todos los paquetes.*



### Observación de ingeniería de software 24.7

*Cuando utilice instrucciones **`import`**, debe especificar instrucciones **`import`** separadas para cada paquete utilizado en el programa.*



**Error común de programación 24.14**

*Asumir que una instrucción* **`import`** *para un paquete completo (por ejemplo,* **`java.awt.*`***) también importa las clases de los subdirectorios de dicho paquete (por ejemplo,* **`java.awt.event.*`***), provoca errores de sintaxis para las clases de los subdirectorios. Debe haber un* **`import`** *separado para cada paquete del que se utilizan las clases.*

Recuerde que los applets heredan de la clase **`JApplet`**, de modo que contienen todos los métodos requeridos por el **`appletviewer`** o el navegador para ejecutar el applet. La línea 6

```
public class AppletSuma extends JApplet {
```

inicia la definición de la clase **`AppletSuma`** e indica que hereda de **`JApplet`**.

Todas las definiciones de las clases inician con una llave izquierda de apertura (fin de la línea 6), **`{`**, y terminan con una llave derecha de cierre, **`}`**, (Línea 40).



**Error común de programación 24.15**

*Si las llaves no se encuentran como pares coincidentes, el compilador indica un error de sintaxis.*

La línea 7

```
double suma;   // suma de los valores introducidos por el usuario
```

es una *declaración de variable de instancia*; cada instancia (objeto) de la clase contiene una copia de cada variable de instancia. Por ejemplo, si existen 10 instancias en ejecución de este objeto, cada instancia contiene su propia copia de **`suma`**. Además, existirán 10 copias distintas de **`suma`** (una para cada applet). Las variables de instancia se declaran en el cuerpo de la definición de la clase, pero no en el cuerpo de cualquier método de la definición de la clase. La declaración anterior establece que **`suma`** es una variable del tipo primitivo **`double`**.

Uno de los beneficios importantes de las variables de instancia es que sus identificadores pueden utilizarse a través de la definición de la clase (es decir, en todos los métodos de la case). Hasta ahora, declaramos todas las variables en el método **`main`** de una aplicación. A las variables definidas en el cuerpo de un método se les conoce como *variables locales* y solamente pueden utilizarse en el cuerpo del método en el que se definen. Otra diferencia entre las variables de instancia y las variables locales es que a las variables de instancia siempre se les asigna un valor predeterminado y a las variables locales no. La variable **`suma`** se inicializa automáticamente en 0.0, debido a que es una variable de instancia.



**Error común de programación 24.16**

*Utilizar una variable local no inicializada, es un error de sintaxis. A todas las variables locales se les debe asignar un valor antes de intentar utilizar el valor de dicha variable.*



**Buena práctica de programación 24.17**

*Inicializar las variables de instancia en lugar de confiar en la inicialización automática, mejora la legibilidad del programa.*

Este applet contiene dos métodos: **`init`** (definición en las líneas 9 a 32) y **`paint`** (definición en las líneas 34 a 39). El método **`init`** es un método especial del applet que por lo general es el primer método definido por el programador en un applet, y con certeza es el primer método del applet en ejecutarse. El método **`init`** se llama una vez durante la ejecución del applet. Por lo general el método *inicializa* las variables de instancia del applet (si requieren inicializarse en un valor diferente de su valor predeterminado), y realiza cualquier tarea una vez que inicia la ejecución del applet.



**Observación de ingeniería de software 24.8**

*El orden en que se definen los métodos en la definición de una clase no tiene efecto en cuanto al orden en el que se llaman en tiempo de ejecución.*

La primera línea en el método **`init`** siempre aparece como

```
public void init()
```

la cual indica que **`init`** es un método público que no devuelve información (**`void`**) cuando completa su tarea, y no recibe argumentos (paréntesis vacíos después de **`init`**).

La llave izquierda (línea 10) marca el inicio del cuerpo de **init**, y la llave derecha correspondiente (línea 32) marca el final de **init**. Las líneas 11 y 12

```
String primerNumero,      // primera cadena introducida por el usuario
       segundoNumero;     // segunda cadena introducida por el usuario
```

son la declaración para las variables locales **primerNumero** y **segundoNumero** de tipo **String**.

Las líneas 13 y 14

```
double numero1,           // primer número a sumar
       numero2,           // segundo número a sumar
```

declaran que las variables **numero1** y **numero2** son tipos de datos primitivos **double**, lo cual significa que estas variables almacenan valores de punto flotante. Éstas son variables de instancia, de modo que se inicializan automáticamente en 0.0 (el valor predeterminado para las variables de instancia **double**).

Observe que en realidad existen dos tipos de variables en Java, *variables de tipos de datos primitivos* (por lo general llamadas *variables*) y *variables de referencia* (por lo general llamadas *referencias*). Los identificadores **primerNumero** y **segundoNumero** son en realidad referencias; nombres utilizados para hacer *referencia a objetos* en el programa. Dichas referencias en realidad contiene la ubicación de un objeto dentro de la memoria de la computadora. En los applets anteriores, el método **paint** en realidad recibe una referencia llamada **g** que hace referencia al objeto **Graphics**. La referencia se utiliza para enviar mensajes al (es decir, llamar métodos de) objeto **Graphics** en memoria que nos permite dibujar sobre un applet. Por ejemplo, la instrucción

```
g.drawString( “Bienvenido a la programacion en Java!”, 25, 25 );
```

envía el mensaje **drawString** (llama al método **drawString**) al objeto **Graphics** al que hace referencia. Como parte del mensaje (llamada al método), proporcionamos los datos que requiere **drawString** para llevar a cabo su tarea. El objeto **Graphics** después dibuja el **String** en la ubicación especificada.

Los identificadores **numero1**, **numero2** y **suma** son los nombres de las *variables*. Una variable es similar a un objeto. La principal diferencia entre una variable y un objeto es que un objeto se define mediante la definición de una clase que puede contener tanto datos (variables de instancia) como métodos, mientras que una variable se define mediante un *tipo de dato primitivo (o predefinido)* (de tipo **char**, **byte**, **short**, **int**, **long**, **float**, **double** o **boolean**) que sólo puede contener datos. Una variable puede almacenar exactamente un valor a la vez, mientras que un objeto puede contener muchas piezas individuales de datos. La diferencia entre una variable y una referencia se basa en el tipo de dato del identificador (como se establece en la declaración). Si el tipo de dato es un nombre de clase, el identificador es una referencia a un objeto y dicha referencia puede utilizarse para enviar mensajes al objeto (llamar a los métodos). Si el tipo de dato es uno de los tipos de datos primitivos, el identificador es una variable que puede utilizarse para almacenar en memoria o para recuperar desde la memoria un valor individual del tipo de dato primitivo declarado.

### Observación de ingeniería de software 24.9

*Una pista para ayudarle a determinar si un identificador es una variable o una referencia es el tipo de dato de la variable. Por convención, todas las clases en Java comienzan con una letra mayúscula. Por lo tanto, si el tipo de dato comienza con una letra mayúscula, por lo general usted puede asumir que el identificador es una referencia a un objeto del tipo declarado (por ejemplo,* ***Graphics g*** *indica que* ***g*** *es una referencia a un objeto* ***Graphics****).*

Las líneas 16 a 19

```
// lee el primer número del usuario
primerNumero =
   JOptionPane.showInputDialog(
      “Introduzca el primer valor de punto flotante” );
```

leen el primer número de punto flotante del usuario. El método **JOptionPane.showInputDialogo** despliega un diálogo de entrada que indica al usuario que introduzca un valor. El usuario escribe un valor en el campo de texto del diálogo de entrada, y luego hace clic en el botón **Aceptar** para devolver la cadena que escribió. [Si usted escribe y no aparece cosa alguna en el campo de texto, coloque el apuntador del ratón en el campo de texto y haga clic para activar el campo de texto.]

Técnicamente, el usuario puede digitar cualquier cosa que desee. En este programa, si el usuario escribe un valor no numérico o hace clic en el botón **Cancelar**, ocurrirá un error en tiempo de ejecución y el mensaje se desplegará en la ventana de comando desde la que se ejecuta el **appletviewer**.

A la variable **primerNumero** se le asigna el resultado de la llamada a la operación **JOptionPane.showInputDialog** en la instrucción de asignación. La instrucción se lee como "**primerNumero** *obtiene* el valor de **JOptionPane.showInputDialog( "Introduzca el primer valor de punto flotante" )**".

Las líneas 21 a 24

```
// lee el segundo número del usuario
segundoNumero =
   JOptionPane.showInputDialog(
      "Introduzca el segundo valor de punto flotante" );
```

lee el segundo valor de punto flotante del usuario, desplegando un cuadro de entrada.

Las líneas 26 a 28

```
// convierte los números del tipo String a tipo double
numero1 = Double.parseDouble( primerNumero );
numero2 = Double.parseDouble( segundoNumero );
```

convierte las dos cadenas de entrada del usuario a valores **double** que puede utilizarse en un cálculo. El método ***Double.parseDouble*** (un método **static** de la clase **Double**) convierte su argumento **String** al valor **double** de punto flotante **Double** que es parte del paquete **java.lang**. El valor de punto flotante devuelto por **Double.parseDouble** en la línea 27 se asigna a la variable **numero1**. Cualquier referencia subsiguiente a **numero1** en el método utiliza este mismo valor de punto flotante. El valor de punto flotante devuelto por **Double.parseDouble** en la línea 28 se asigna a la variable **numero2**. Cualquier referencia subsiguiente a **numero2** en el método utiliza este valor de punto flotante.

### Observación de ingeniería de software 24.10

*Para cada tipo de dato primitivo (tal como un **int** o un **double**) existe una clase correspondiente (tal como **Integer** o **Double**) en el paquete **java.lang**. Estas clases (por lo general conocidas como envolturas de tipo) proporcionan métodos para procesar valores de tipos de datos primitivos (tales como convertir un **String** a un valor de tipo de dato primitivo, o convertir un valor de tipo de dato primitivo a un **String**). Los tipos de datos primitivos no tienen métodos. Por lo tanto, los métodos relacionados con un tipo de dato primitivo se ubican en la clase envolvente de tipo correspondiente (es decir, el método **parseDouble** que convierte un **String** a un valor **double** se localiza en la clase **Double**).*

La instrucción de asignación de la línea 31

```
suma = numero1 + numero2;
```

calcula la suma de las variables **numero1** y **numero2** y asigna el resultado a la variable **suma** por medio del operador =. La instrucción se lee como "**suma** *obtiene* el valor de **numero1** + **numero2**". La mayoría de los cálculos se realizan con instrucciones de asignación. Observe que la variable de instancia **suma** se utiliza en la instrucción anterior en el método **init**, aun cuando **suma** no se definió en el método **init**. Definimos **suma** como una variable de instancia, por lo que podemos utilizar **init** y todos los otros métodos de la clase.

El método **init** del applet retorna y el **appletviewer** o el navegador llama al método **start** del applet. No definimos el método **start** en este applet, por lo que aquí utilizamos el método proporcionado por la clase **JApplet**. El método **start** se utiliza primordialmente con un concepto avanzado llamado subprocesamiento múltiple, el cual no explicamos en estos capítulos introductorios.

A continuación, el navegador llama al método **paint** del applet. En este ejemplo, el método **paint** dibuja un rectángulo que contiene una cadena con el resultado de la suma. La línea 37

```
g.drawRect( 15, 10, 270, 20);
```

envía el mensaje **drawRect** al objeto **Graphics** al que **g** hace referencia (llama al método **drawRect** del objeto **Graphics**). El método **drawRect** dibuja un rectángulo, basándose en sus cuatro argumentos. Los primeros dos valores enteros representan la *coordenada superior izquierda x*, y la *coordenada superior izquierda y*, en donde el objeto **Graphics** comienza el dibujo del rectángulo. El tercero y cuarto argumentos son núme-

ros enteros negativos que representan el *ancho* y la *altura* del rectángulo en pixeles, respectivamente. Esta instrucción en particular dibuja un rectángulo que comienza con la coordenada (*15*, *10*) que es de 270 pixeles de ancho y de 20 pixeles de alto.



**Error común de programación 24.17**

*Proporcionar un ancho negativo o una altura negativa como argumentos del método* **`drawRect`** *de* **`Graphics`**, *es un error lógico. El rectángulo no se desplegará y no indicará error alguno.*



**Error común de programación 24.18**

*Proporcionar dos puntos (es decir, pares de coordenadas* x *y* y*) como argumentos del método* **`drawRect`** *de* **`Graphics`**, *es un error lógico. El tercer argumento debe ser el ancho en pixeles y el cuarto argumento debe ser la altura en pixeles del rectángulo a dibujar.*



**Error común de programación 24.19**

*Por lo general, proporcionar argumentos al método* **`drawRect`** *de* **`Graphics`** *que provoquen que el rectángulo se dibuje fuera del área visible del applet (es decir, el ancho y la altura del applet como se especifica el documento HTML que hace referencia al applet), es un error lógico. Aumente el ancho y la altura del applet en el documento HTML, o pase los argumentos al método* **`drawRect`** *que provoca que el rectángulo se dibuje dentro del área visible del applet.*

La línea 38

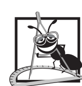
```
g.drawString( "La suma es " + suma, 25, 25 );
```

envía el mensaje **`drawString`** al objeto **`Graphics`** al cual hace referencia **`g`** (llama al método **`drawString`** del objeto **`Graphics`**). La expresión

```
"La suma es " + suma
```

de la instrucción anterior utiliza el operador de concatenación + para concatenar la cadena **`"La suma es "`** y **`suma`** (convertida a una cadena) para crear la cadena desplegada por **`drawString`**. Observe nuevamente que la variable de instancia **`suma`** de la instrucción anterior se utiliza, incluso si no se define en el método **`paint`**.

El beneficio de definir **`suma`** como una variable de instancia es que pudimos asignar a **`suma`** un valor en **`init`** y utilizar el valor **`suma`** en el método **`paint`** más adelante en el programa. Todos los métodos de la clase son capaces de utilizar las variables de instancia en la definición de la clase.



**Observación de ingeniería de software 24.11**

*Las únicas instrucciones que deben colocarse en el método* **`init`** *del applet son aquellas que se relacionan directamente con la inicialización única de las variables de instancia del applet. Los resultados del applet deben desplegarse a través de otros métodos de la clase del applet. Los resultados que involucran el dibujo deben desplegarse desde el método* **`paint`** *del applet.*



**Observación de ingeniería de software 24.12**

*Las únicas instrucciones que deben colocarse en el método* **`paint`** *del applet son aquellas que se relacionan de manera directa con el dibujo (es decir, las llamadas a los métodos de la clase* **`Graphics`***) y con la lógica del dibujo. Por lo general, los cuadros de diálogo no deben desplegarse desde el método* **`paint`** *del applet.*

En este capítulo introdujimos muchas características importantes de Java, que incluyen aplicaciones, applets, el desplegado de datos en la pantalla, la entrada de datos desde el teclado, la realización de cálculos y la toma de decisiones. En el siguiente capítulo explicaremos algunas de las diferencias entre Java y C/C++, tal como arreglos, operadores y definiciones de métodos. En los siguientes capítulos explicaremos la programación basada en objetos y orientada a objetos, así como los gráficos en Java, interfaces gráficas de usuario (GUIs) y características multimedia.

## RESUMEN

- Java es uno de los lenguajes de desarrollo más populares en la actualidad.
- Java fue desarrollado en Sun Microsystems. Sun proporciona una implementación de la plataforma de Java llamada Java 2 Software Development Kit (J2SDK), la cual incluye el conjunto mínimo de herramientas necesarias para escribir programas en Java.

- Java es un lenguaje completamente orientado a objetos con un enorme soporte para las técnicas de ingeniería de software.
- Por lo general, los sistemas en Java constan de varias partes: el lenguaje, la Interfaz de Programación de Aplicaciones de Java (API, Java Applications Programming Interface), y distintas bibliotecas de clases.
- Por lo general, los programas en Java pasan a través de cinco etapas para poder ejecutarse: edición, compilación, carga, verificación y ejecución.
- Los nombres de programas en Java terminan con la extensión **.java**.
- El compilador de Java (**javac**) traduce un programa en Java a bytecodes, el código comprensible para el intérprete de Java. Si un programa se compila correctamente, se produce un archivo con extensión **.class**. Este archivo contiene los bytecodes que se interpretan durante la fase de ejecución.
- Un programa Java primero debe colocarse en memoria, antes de que pueda ejecutarse. Esto se hace por medio del cargador de clases, el cual toma el archivo (o archivos) **.class** que contienen los bytecodes y los transfiere a la memoria. El archivo **.class** puede cargarse desde un disco en su sistema o sobre una red.
- Una aplicación es un programa que se ejecuta por medio del intérprete **java**.
- Un comentario que comienza con // se llama comentario de una sola línea. Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad.
- A una cadena de caracteres contenida entre comillas se le llama cadena, cadena de caracteres, mensaje, o literal de cadena.
- La palabra reservada **class** introduce la definición de una clase y de inmediato le sigue el nombre de la clase.
- Las palabras reservadas (o palabras clave) están reservadas para el uso de Java.
- Por convención, todos los nombres de clases en Java comienzan con una letra mayúscula. Si el nombre de una clase tiene más de una palabra, cada palabra debe comenzar con mayúscula.
- Un identificador consiste en una serie de caracteres que consta de letras, dígitos, guiones bajos ( _ ) y signos de moneda (**$**) que no comienza con un dígito, no contiene espacios y no es una palabra reservada.
- Java es sensible a mayúsculas y a minúsculas, es decir, las letras mayúsculas y minúsculas son diferentes.
- Una llave izquierda, **{**, inicia el cuerpo de la definición de una clase. Su correspondiente llave derecha, **}**, finaliza la definición de la clase.
- Las aplicaciones en Java comienzan su ejecución en el método **main**.
- La primera línea del método **main** debe definirse como:

```
public static void main( String args[] )
```

- Una llave izquierda, **{**, comienza el cuerpo de la definición de un método. Su correspondiente llave derecha, **}**, termina el cuerpo de la definición del método.
- A **System.out** se le conoce como el objeto estándar de salida. **System.out** permite a las aplicaciones de Java desplegar cadenas y otro tipo de información en la ventana de comando desde la cual se ejecuta la aplicación Java.
- La secuencia de escape **\n** significa nueva línea. Otras secuencias de escape incluyen **\t** (tabulador), **\r** (retorno de carro), **** (diagonal invertida) y **\"** (comillas dobles).
- El método **println** del objeto **System.out** despliega (o imprime) una línea de información en la ventana de comandos. Cuando **println** completa su tarea, el cursor se posiciona automáticamente al principio de la siguiente línea en la ventana de comando.
- Toda instrucción debe terminar con punto y coma (también conocido como terminador de instrucción).
- La diferencia entre **System.out.print** y **System.out.println** es que **System.out.print** no posiciona el cursor al principio de la siguiente línea en la ventana de comando cuando termina de desplegar su argumento. El siguiente carácter que se despliega en la ventana de comando aparecerá inmediatamente después del último carácter desplegado con **System.out.print**.
- Java contiene muchas clases predefinidas que se agrupan en directorios del disco, dentro de categorías de clases relacionadas llamadas paquetes. A los paquetes en su conjunto se les conoce como la biblioteca de clases de Java o la interfaz de programación de aplicaciones de Java (API de Java).
- La clase **JOptionPane** está definida para nosotros en un paquete llamado **javax.swing**. La clase **jOptionPane** contiene métodos que despliegan un cuadro de diálogo que contiene información.
- El compilador utiliza instrucciones **import** para localizar las clases requeridas para compilar un programa en Java.
- El paquete **javax.swing** contiene muchas clases que ayudan a definir interfaces gráficas de usuario (GUI) para una aplicación. Los componentes GUI facilitan la entrada de datos del usuario y la salida de datos del programa.

- El método **showMessageDialog** de la clase **JOptionPane** requiere dos argumentos. Hasta que expliquemos **JOptionPane** con detalle en el capítulo 29, el primer argumento siempre será la palabra reservada **null**. El segundo argumento es la cadena a desplegar.
- Se llama a un método estático al colocar a continuación del nombre de la clase un punto (**.**) y el nombre del método.
- El método **exit** de la clase **System** finaliza una aplicación. La clase **System** es parte del paquete **java.lang**. El paquete **java.lang** se importa automáticamente en todos los programas Java.
- Las variables de tipo **int** almacenan valores de tipo entero (es decir, números completos tales como 7, −11, 0, 31914).
- A los tipos tales como **int**, **float**, **double** y **char** con frecuencia se les llama tipos de datos primitivos. Los nombres de tipos primitivos son palabras reservadas del lenguaje de programación Java.
- El método **Integer.parseInt** (un método estático de la clase **Integer**) convierte su argumento de tipo cadena a un entero.
- Java tiene una versión del operador **+** para la concatenación de cadenas que permite concatenar una cadena y un valor de otro tipo de dato (incluso otra cadena).
- Los nombres de variables corresponden a ubicaciones en la memoria de la computadora. Toda variable tiene un nombre, un tipo, un tamaño y un valor.
- Toda variable declarada en un método debe inicializarse antes de que pueda utilizarse en una expresión.
- Un applet es un programa en Java que puede ejecutarse en el **appletviewer** (una utilidad de prueba para applets que se incluye con J2SDK), o en un navegador de la World Wide Web como Netscape Communicator o el Internet Explorer de Microsoft. El **appletviewer** (o el navegador) ejecuta un applet cuando un documento en Lenguaje de Marcación de Hipertexto (HTML) que contiene un applet se abre en el **appletviewer** (o en el navegador).
- En el **appletviewer**, usted puede ejecutar de nuevo un applet, haciendo clic en el menú **Subprograma** y seleccionando la opción **Volver a cargar**. Para finalizar un applet, haga clic en el menú **Subprograma** del **appletviewer** y seleccione la opción **Salir**.
- La clase **Graphics** se encuentra localizada en el paquete **java.awt**. Importe la clase **Graphics** para que el programa pueda dibujar gráficos.
- La clase **JApplet** se localiza en el paquete **javax.swing**. Cuando crea un applet en Java, por lo general se importa la clase **JApplet**.
- Cada porción del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en el API de Java se almacenan en el directorio **java** o **javax**, el cual contiene muchos subdirectorios.
- Java utiliza la herencia para crear nuevas clases a partir de definiciones de clases existentes. La palabra reservada **extends**, seguida por el nombre de la clase, indica la clase a partir de la cual hereda una nueva clase.
- En la relación de herencia, a la clase a continuación de **extends** se le llama superclase o clase base, y a la nueva clase se le llama subclase o clase derivada. El uso de la herencia da como resultado una nueva definición de clase que tiene los atributos (datos) y los comportamientos (métodos) de la superclase, así como las nuevas características adicionadas en la definición de la subclase.
- Uno de los beneficios de extender la clase **JApplet** es que alguien más ya definió lo que "significa un applet". El **appletviewer** y los navegadores de la World Wide Web que soportan applets esperan que cada applet de Java contenga ciertas capacidades (atributos y comportamientos), y la clase **JApplet** ya proporciona todas esas capacidades.
- Las clases se utilizan como "plantillas" o "anteproyectos" para instanciar (o crear) objetos en memoria a utilizarse dentro de un programa. Un objeto (o instancia) es una región en la memoria de la computadora en la cual la información se almacena para utilizarse en un programa. Por lo general, el término objeto implica que los atributos (datos) y los comportamientos (métodos) se asocian con el objeto, y que dichos comportamientos realizan operaciones en los atributos del objeto.
- El método **paint** es uno de los tres métodos (comportamientos) que seguramente se invocarán automáticamente cuando inicie la ejecución de cualquier applet. Estos tres métodos son **init**, **start** y **paint**, y con seguridad se llamarán en ese orden. Estos métodos se llaman desde el **appletviewer** o desde el navegador en el que se ejecuta el applet.
- El método **drawString** de la clase **Graphics** dibuja una cadena en una ubicación específica del applet. El primer argumento para **drawString** es la **String** (cadena) a dibujar. Los dos últimos argumentos en la lista son las coordenadas (o posiciones) en las cuales se debe dibujar una cadena. Las coordenadas se miden en pixeles desde la esquina superior izquierda del applet.
- Usted debe crear un archivo HTML (Lenguaje de Marcación de Hipertexto) para cargar un applet dentro del **appletviewer** (o navegador) para ejecutarlo.

- Muchos códigos HTML (conocidos como etiquetas) vienen en pares. Las etiquetas de HTML comienzan con una llave angular izquierda **<**, y terminan con una llave angular derecha **>**.
- Por lo general, el applet y su correspondiente archivo HTML se almacenan en el mismo directorio del disco.
- El primer componente de la etiqueta **<applet>** indica el archivo que contiene la clase con el applet compilado. El segundo y el tercer componente de la etiqueta **<applet>** indica el ancho (**width**) y la altura (**height**) del applet en pixeles. Por lo general, cada applet debe ser menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones como ancho y altura mínimos).
- El **appletviewer** solamente comprende las etiquetas **<applet>** y **</applet>** de HTML, de modo que en ocasiones se le llama "navegador mínimo" (ignora todas las demás etiquetas de HTML).
- El método **drawLine** de la clase **Graphics** dibuja líneas. El método requiere cuatro argumentos que representan los dos puntos extremos de la línea en un applet, la coordenada *x* y la coordenada *y* del primer punto extremo de la línea, y la coordenada *x* y la coordenada *y* del segundo punto extremo de la línea. Todos los valores de las coordenadas se especifican con respecto a la coordenada superior izquierda (0, 0) del applet.
- El tipo de dato primitivo **double** almacena números de punto flotante de doble precisión. El tipo de dato primitivo **float** almacena números de punto flotante de precisión simple. Un **double** requiere más memoria de almacenamiento que un valor de punto flotante, pero lo almacena con aproximadamente el doble de precisión que un **float** (15 dígitos significativos para **double** contra siete dígitos significativos para **float**).
- Las instrucciones **import** no son necesarias, si usted siempre utiliza el nombre completo de una clase, incluyendo el nombre completo del paquete y el nombre de la clase.
- La notación asterisco (*****) después del nombre de un paquete en un **import** indican que todas las clases del paquete deben estar disponibles para el compilador, de modo que éste pueda asegurarse de que las clases utilizan de manera correcta. Esto permite a los programadores utilizar el nombre corto de cualquier clase desde el paquete en el programa.
- Cada instancia (objeto) de una clase contiene una copia de cada variable de instancia. Las variables de instancia se declaran en el cuerpo de la definición de la clase, pero no en el cuerpo de cualquier método de la definición de la clase. Un beneficio importante de las variables de instancia es que sus identificadores pueden utilizarse a través de la definición de la clase (es decir, en todos sus métodos).
- De nuevo, durante la ejecución del applet se llama al método **init**. Por lo general, este método inicializa las variables de instancia del applet y realiza cualquiera de las tareas que necesitan llevarse a cabo una vez, al inicio de la ejecución del applet.
- El método **Double.parseDouble** (un método estático de la clase **Double**) convierte su argumento **String** en un valor de punto flotante. La clase **Double** es parte del paquete **java.lang**.

## TERMINOLOGÍA

**!=** "no es igual que "
**<** "es menor que"
**<=** "es menor o igual que"
**==** "es igual que"
**>** "es mayor que"
**>=** "es mayor o igual que"
aplicación
applet
argumento de un método
asociatividad de derecha a izquierda
asociatividad de los operadores
barra de título de un diálogo
biblioteca de clases de Java
cadena
cadena de caracteres
cadena vacía (**""**)
carácter de escape diagonal invertida (****)
carácter de nueva línea (**\n**)
caracteres blancos
clase
clase definida por el programador
clase definida por el usuario
clase **Integer**
clase **JOptionPane**
clase **String**
clase **System**
comentario (**//**)
comentario de documentación de Java
comentario de una sola línea
comentario de varias líneas
compilador
concatenación de cadenas
cuadro de diálogo
cuerpo de la definición de un método
cuerpo de la definición de una clase
cursor del ratón
declaración
definición de una clase
diálogo de entrada
diálogo de mensaje
división entera
documentar un programa
entero (**int**)
error de compilación
error de sintaxis
error del compilador
error en tiempo de compilación
extensión de archivo **.class**
extensión de archivo **.java**
**false**
forma en línea recta
herramienta de comando
herramienta shell
identificador
indicador
indicador de MS-DOS
instrucción
instrucción de asignación
instrucción **import**

interfaz de programación de aplicaciones (API) de Java
interfaz gráfica de usuario (GUI)
intérprete
intérprete **java**
Java
Java 2 Software Development Kit (J2SDK)
**JOptionPane.ERROR_MESSAGE**
**JOptionPane.INFORMATION_MESSAGE**
**JOptionPane.PLAIN_MESSAGE**
**JOptionPane.QUESTION_MESSAGE**
**JOptionPane.showInputDialog**
**JOptionPane.showMessageDialog**
**JOptionPane.WARNING_MESSAGE**
la llave derecha **}** termina el cuerpo de un método
la llave derecha **}** termina el cuerpo de una clase
la llave izquierda **{** comienza el cuerpo de un método
la llave izquierda **{** comienza el cuerpo de una clase
lista separada por comas
literal
llaves (**{ y }**)
memoria
mensaje
método
método **main**
método **parseInt** de la clase **Integer**
método **static**
método **System.exit**
método **System.out.print**
método **System.out.println**
navegador Internet Explorer de Microsoft
navegador Netscape Communicator
nombre de una clase
nombre de variable
objeto
objeto de salida estándar
operador
operador binario
operador de asignación (**=**)
operador de concatenación de cadenas (**+**)
operador de división (**/**)
operador de multiplicación (*****)
operador de suma (**+**)
operador de sustracción (**-**)
operador módulo (**%**)
operadores de igualdad
operadores de relación
operando
palabra reservada **class**
palabra reservada **public**
palabra reservada **void**
palabras reservadas
paquete
paquete **java.lang**
paquete **java.swing**
paréntesis **()**
paréntesis anidados
precedencia
puntero del ratón
reglas de precedencia de operadores
secuencia de escape
sensible a mayúsculas y minúsculas
**System.out**
terminador de instrucción (**;**)
terminador de instrucción punto y coma (**;**)
tipo de dato primitivo
tipo primitivo **int**
**true**
ubicación de memoria
valor de variable
variable
ventana de comando

## *ERRORES COMUNES DE PROGRAMACIÓN*

**24.1** Los errores como la división entre cero ocurren durante la ejecución del programa, de modo que estos errores se llaman errores en tiempo de ejecución o errores de ejecución. Los errores fatales en tiempo de ejecución provocan que los programas terminen de inmediato, sin tener éxito al realizar sus tareas. Los errores no fatales en tiempo de ejecución permiten a los programas completar su ejecución, por lo general con resultados incorrectos.

**24.2** Olvidar uno de los delimitadores de un comentario de varias líneas, es un error de sintaxis.

**24.3** Java es sensible a mayúsculas y minúsculas. Por lo general, no utilizar las letras mayúsculas y minúsculas apropiadas para un identificador, es un error de sintaxis.

**24.4** Para una clase pública, es un error si el nombre de archivo no es idéntico al nombre de la clase tanto en las letras, como en las mayúsculas y las minúsculas. Por lo tanto, también es un error que un archivo contenga dos o más clases públicas.

**24.5** Es un error no finalizar el nombre de un archivo con la extensión **.java**, si contiene la definición una clase de la aplicación. El compilador de Java no podrá compilar la definición de la clase.

**24.6** Si las llaves no están en pares coincidentes, el compilador indica un error.

**24.7** Omitir el punto y coma al final de una instrucción, es un error de sintaxis.

**24.8** Dividir una instrucción a la mitad de un identificador o de una cadena, es un error de sintaxis.

**24.9** Olvidar llamar a **System.exit** en una aplicación que despliega una interfaz gráfica, evita que el programa termine de manera apropiada. Por lo general, esto provoca que no sea posible introducir comando alguno.

**24.10** Confundir el operador **+** utilizado para la concatenación de cadenas con el operador **+** utilizado para la suma puede provocar resultados extraños. Por ejemplo, al asumir que la variable entera **y** tiene el valor **5**, la expresión **"y + 2 = " + y + 2** arroja como resultado la cadena **"y + 2 = 52"**, no **"y + 2 = 7"**, debido a que el primer valor de **y** se concatena con la cadena **"y + 2 ="**, después el valor **2** se concatena con la cadena más grande **"y + = 5"**. La expresión **"y + 2 =" + (y + 2)** produce el resultado deseado.

**24.11** Colocar caracteres adicionales tales como comas (`,`) entre los componentes de la etiqueta `<applet>` puede provocar que el `appletviewer` o el navegador produzcan un mensaje de error que indica un `MissingResourceException` al cargar el applet.

**24.12** Olvidar la etiqueta `</applet>` provoca la carga incorrecta del applet dentro del `appletviewer` o el navegador.

**24.13** Ejecutar el `appletviewer` con un nombre de archivo que no termina con `.html` o `.htm` provoca un error que evita que el `appletviewer` cargue su applet para ejecución.

**24.14** Asumir que una instrucción `import` para un paquete completo (por ejemplo, `java.awt.*`) también importa las clases de los subdirectorios de dicho paquete (por ejemplo, `java.awt.event.*`), provoca errores de sintaxis para las clases de los subdirectorios. Debe haber un `import` separado para cada paquete del que se utilizan las clases.

**24.15** Si las llaves no se encuentran como pares coincidentes, el compilador indica un error de sintaxis.

**24.16** Utilizar una variable local no inicializada, es un error de sintaxis. A todas las variables locales se les debe asignar un valor antes de intentar utilizar el valor de dicha variable.

**24.17** Proporcionar un ancho negativo o una altura negativa como argumentos del método `drawRect` de `Graphics`, es un error lógico. El rectángulo no se desplegará y no indicará error alguno.

**24.18** Proporcionar dos puntos (es decir, pares de coordenadas *x* y *y*) como argumentos del método `drawRect` de `Graphics`, es un error lógico. El tercer argumento debe ser el ancho en pixeles y el cuarto argumento debe ser la altura en pixeles del rectángulo a dibujar.

**24.19** Por lo general, proporcionar argumentos al método `drawRect` de `Graphics` que provoquen que el rectángulo se dibuje fuera del área visible del applet (es decir, el ancho y la altura del applet como se especifica el documento HTML que hace referencia al applet), es un error lógico. Aumente el ancho y la altura del applet en el documento HTML, o pase los argumentos al método `drawRect` que provoca que el rectángulo se dibuje dentro del área visible del applet.

## TIPS PARA PREVENIR ERRORES

**24.1** Siempre pruebe los programas en Java en todos los sistemas en los que desee ejecutarlos.

**24.2** Cuando el compilador reporta un error de sintaxis, el error podría no estar en la línea que indica el mensaje de error. Primero, verifique la línea en donde se reporta el error. Si la línea no contiene errores de sintaxis, verifique las líneas anteriores del programa.

**24.3** Si el comando `appletviewer` no funciona y/o el sistema indica que el comando `appletviewer` no se encuentra, la variable de ambiente `PATH` podría no estar definida apropiadamente en su computadora. Revise las direcciones de instalación para el Java 2 Software Development Kit para asegurarse de que la variable de ambiente está correctamente definida para su sistema (en algunas computadoras, podría ser necesario reiniciar el equipo después de definir la variable de ambiente `PATH`).

**24.4** El mensaje de error del compilador "Public class ClassName must be defined in a file called *ClassName*.java" indica que 1) el nombre del archivo no coincide exactamente con el nombre de la clase `public` en el archivo (incluidas todas las letras mayúsculas y minúsculas), o 2) que usted escribió el nombre de la clase de manera incorrecta cuando compiló la clase (el nombre debe deletrearse con las letras mayúsculas y minúsculas apropiadas).

**24.5** Si usted recibe un mensaje de error `MissingResourceException` durante la carga de un applet dentro del `appletviewer` o del navegador, verifique cuidadosamente la etiqueta `<applet>` en el archivo HTML para ver si hay de errores de sintaxis. Compare su archivo HTML con el archivo de la figura 24.15 para confirmar una sintaxis adecuada.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**24.1** Escriba sus programas en Java de manera sencilla y directa. A esto en ocasiones se le llama KIS ("keep it simple", "manténgalo simple". No deshaga el lenguaje, intentado usos extraños.

**24.2** Lea la documentación para la versión de Java que va a utilizar. Consulte esta documentación con frecuencia para asegurarse de que conoce la rica colección de características de Java y de que utiliza correctamente estas características.

**24.3** Su computadora y su compilador son buenos maestros. Si después de leer cuidadosamente el manual de la documentación de Java no está seguro de la manera en que funciona una característica de Java, experimente y vea qué sucede. Estudie cada mensaje de error o de advertencia que obtenga cuando compile sus programas, y corríjalos para eliminar dichos mensajes.

**24.4** Por convención, usted siempre debe comenzar el nombre de una clase con la primera letra en mayúscula.

**24.5** Cuando lea un programa en Java, busque identificadores que comiencen con la primera letra en mayúscula. Por lo general, éstos representan clases de Java.

**24.6** Cada vez que introduzca una llave izquierda de apertura, **`{`**, en su programa, introduzca inmediatamente la llave derecha de cierre, **`}`**, y vuelva a colocar el indicador entre las llaves para comenzar a introducir el cuerpo del programa. Esto ayuda a evitar que falten llaves.

**24.7** Sangre el cuerpo entero de cada definición de clase un "nivel" entre la llave izquierda, **`{`**, y la llave derecha, **`}`**, que define el cuerpo de la clase. Esto enfatiza la estructura de la definición de la clase, y ayuda a que las definiciones de clases sean más fáciles de leer.

**24.8** Establezca una convención para el tamaño del sangrado que prefiera, y entonces aplique de manera uniforme dicha convención. Puede utilizar la tecla tab para crear el sangrado, aunque tab podría variar entre editores. Le recomendamos el uso de tabuladores de 1/4 de pulgada o (preferiblemente) tres espacios para formar un nivel de sangrado.

**24.9** Sangre por completo el cuerpo de cada definición de método un "nivel" entre la llave izquierda, **`{`**, y la llave derecha, **`}`**. Esto hace que la estructura del método resalte, y ayuda a que la definición del método sea más fácil de leer.

**24.10** Coloque un espacio después de cada coma (**`,`**) en una lista de argumentos, para hacer más legibles los programas.

**24.11** Elegir nombres de variables significativas (descriptivas) ayuda a un programa a estar "autodocumentado" (es decir, se vuelve más sencillo comprender un programa sólo con leerlo, y no es necesario tener que leer los manuales o utilizar comentarios en exceso).

**24.12** Por convención, los identificadores de nombres de variables comienzan con una letra minúscula. Así como con los nombres de las clases, cada palabra del nombre después de la primera, debe comenzar con una letra mayúscula. Por ejemplo, el identificador **`primerNumero`** tiene una letra mayúscula **`N`** en la segunda palabra **`Numero`**.

**24.13** Algunos programadores prefieren declarar cada variable en una línea aparte. Este formato permite insertar fácilmente un comentario descriptivo después de cada declaración.

**24.14** Coloque espacios de cualquier lado de un operador binario. Esto hace que el operador sobresalga y hace al programa más legible.

**24.15** Investigue cuidadosamente las capacidades de cualquier clase en la documentación del API de Java, antes de heredar a una subclase. Esto ayuda a asegurar que el programador no redefine por descuido una capacidad que ya está proporcionada.

**24.16** Siempre pruebe el applet en el **`appletviewer`** y asegúrese de que se ejecuta correctamente, antes de cargar el applet en un navegador de la World Wide Web. Con frecuencia, los navegadores guardan una copia de un applet en la memoria hasta que termina la sesión actual de navegación (es decir, hasta que se cierran todas las ventanas del navegador). Por lo tanto, si usted modifica un applet, recompílelo, y luego recargue el applet en el navegador; es probable que no vea los cambios, debido a que el navegador aún está ejecutando la versión original del applet. Cierre todas las ventanas de su navegador para eliminar de la memoria la versión anterior del applet. Abra una nueva ventana del navegador y cargue el applet para ver los cambios.

**24.17** Inicializar las variables de instancia en lugar de confiar en la inicialización automática, mejora la legibilidad del programa.

## *TIP DE RENDIMIENTO*

**24.1** Los intérpretes tienen una ventaja sobre los compiladores en el mundo de Java, a saber, que un programa interpretado puede comenzar su ejecución de inmediato, tan pronto como se descarga en la máquina del cliente, mientras que un programa a compilarse primero debe sufrir un retraso potencialmente largo mientras el programa se compila antes de que pueda ejecutarse.

## *TIPS DE PORTABILIDAD*

**24.1** Aunque es más fácil escribir programas portables en Java que en la mayoría de los demás lenguajes de programación, existen diferencias entre los compiladores, los intérpretes y las computadoras que pueden hacer de la portabilidad una meta difícil de alcanzar. El simple hecho de escribir programas en Java, no garantiza la portabilidad. Ocasionalmente el programador necesitará lidiar directamente con las variaciones entre los compiladores y las computadoras.

**24.2** Verifique sus applets en todos los navegadores utilizados por la gente que ve su applet. Esto le ayudará a asegurar que la gente que vea su applet experimente la funcionalidad que usted espera. [*Nota:* Una meta del plug-in de Java (que explicaremos posteriormente) es proporcionar la ejecución consistente de un applet en diferentes navegadores.]

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**24.1** Evite utilizar identificadores que contengan signos de moneda (**$**), ya que con frecuencia el compilador los utiliza para crear nombres de identificadores.

**24.2** Si su navegador Web no soporta Java 2, la mayoría de los applets de este libro no se ejecutarán en su navegador. Esto se debe a que la mayoría de los applets de este libro utilizan las características que son nuevas en Java 2, o a que no se proporcionan con los navegadores que soportan Java 1.1.

**24.3** Por lo general, cada applet debe tener un tamaño menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones de ancho y altura mínimas).

**24.4** El compilador de Java no necesita instrucciones **import** en un archivo de código fuente de Java si el nombre completo de la clase, es decir, el nombre completo del paquete y el nombre de la clase (por ejemplo, **java.awt.Graphics**), se especifica cada vez que se utiliza el nombre de la clase en el código fuente.

**24.5** El compilador no carga cada clase en un paquete cuando encuentra una instrucción **import** que utiliza la notación ***** (por ejemplo, **javax.swing.***) para indicar que se utilizan diversas clases del paquete dentro del programa. El compilador busca en el paquete solamente las clases que utiliza el programa.

**24.6** Muchos directorios de paquetes tienen subdirectorios. Por ejemplo, el directorio del paquete **java.awt** contiene el subdirectorio **event** para el paquete **java.awt.event**. Cuando el compilador encuentra una instrucción **import** que utiliza la notación ***** (por ejemplo, **java.awt.***) para indicar que se utilizan distintas clases del paquete dentro del programa, el compilador no busca el subdirectorio **event**. Esto significa que usted no puede definir un **import** de **java.*** para buscar las clases de todos los paquetes.

**24.7** Cuando utilice instrucciones **import**, debe especificar instrucciones **import** separadas para cada paquete utilizado en el programa.

**24.8** El orden en que se definen los métodos en la definición de una clase no tiene efecto en cuanto al orden en el que se llaman en tiempo de ejecución.

**24.9** Una pista para ayudarle a determinar si un identificador es una variable o una referencia es el tipo de dato de la variable. Por convención, todas las clases en Java comienzan con una letra mayúscula. Por lo tanto, si el tipo de dato comienza con una letra mayúscula, por lo general usted puede asumir que el identificador es una referencia a un objeto del tipo declarado (por ejemplo, **Graphics g** indica que **g** es una referencia a un objeto **Graphics**).

**24.10** Para cada tipo de dato primitivo (tal como un **int** o un **double**) existe una clase correspondiente (tal como **Integer** o **Double**) en el paquete **java.lang**. Estas clases (por lo general conocidas como envolturas de tipo) proporcionan métodos para procesar valores de tipos de datos primitivos (tales como convertir un **String** a un valor de tipo de dato primitivo, o convertir un valor de tipo de dato primitivo a un **String**). Los tipos de datos primitivos no tienen métodos. Por lo tanto, los métodos relacionados con un tipo de dato primitivo se ubican en la clase envolvente de tipo correspondiente (es decir, el método **parseDouble** que convierte un **String** a un valor **double** se localiza en la clase **Double**).

**24.11** Las únicas instrucciones que deben colocarse en el método **init** del applet son aquellas que se relacionan directamente con la inicialización única de las variables de instancia del applet. Los resultados del applet deben desplegarse a través de otros métodos de la clase del applet. Los resultados que involucran el dibujo deben desplegarse desde el método **paint** del applet.

**24.12** Las únicas instrucciones que deben colocarse en el método **paint** del applet son aquellas que se relacionan de manera directa con el dibujo (es decir, las llamadas a los métodos de la clase **Graphics**) y con la lógica del dibujo. Por lo general, los cuadros de diálogo no deben desplegarse desde el método **paint** del applet.

## EJERCICIOS DE AUTOEVALUACIÓN

**24.1** Complete los espacios en blanco:

a) ____________ comienza un comentario de una sola línea.
b) La clase ____________ despliega diálogos de mensaje y diálogos de entrada.
c) Las ____________ están reservadas para el uso de Java.
d) Las aplicaciones Java comienzan su ejecución en el método ____________.
e) Los métodos ____________ y ____________ despliegan información en la ventana de comando.
f) Siempre se llama a un método ____________ usando el nombre de la clase seguido por un punto (**.**) y por el nombre de su método.

**24.2** Diga si es *verdadera* o *falsa* cada una de las siguientes frases. Si es *falsa*, explique por qué.

a) Los comentarios provocan que la computadora imprima en la pantalla el texto que se encuentra después de //, cuando se ejecuta el programa.
b) Al declararse, todas las variables deben tener un tipo.

c) Java considera que las variables **numero** y **NuMero** son idénticas.
d) El método **Integer.parseInt** convierte un entero a una **String**.

**24.3** Escriba instrucciones en Java para llevar a cabo cada una de las siguientes tareas:
a) Declare las variables **c**, **estaEsUnaVariable**, **q76354** y **número** de tipo **int**.
b) Despliegue un diálogo que solicite al usuario que introduzca un entero.
c) Convierta una **String** a un entero y almacene el valor en la variable entera **edad**. Asuma que la cadena se almacena en **valorCadena**.
d) Si la variable **numero** no es igual que **7**, despliegue **"La variable numero no es igual que 7"** en un diálogo de mensaje. [*Pista:* Utilice una versión del diálogo de mensaje que requiere dos argumentos.]
e) Imprima el mensaje **"Este es un programa en Java"** en una línea dentro de la ventana de comandos.
f) Imprima el mensaje **"Este es un programa en Java"** en dos líneas en la ventana de comandos, en donde la primera línea termina con **programa**. Utilice una sola instrucción.

**24.4** Identifique y corrija los errores en la siguiente instrucción:

```
if( c=> 7 )
   JOptionPane.showMessageDialog( null,
   "c es igual o mayor que 7");
```

**24.5** Complete los espacios en blanco.
a) La clase ____________ proporciona métodos para dibujar.
b) Los applets de Java comienzan la ejecución con una serie de tres llamadas a los métodos: ____________, ____________ y ____________.
c) Los métodos ____________ y ____________ despliegan líneas y rectángulos.
d) La palabra reservada ____________ se utiliza para indicar que una nueva clase es una subclase de una clase existente.
e) Todo applet de Java debe extenderse a partir de la clase ____________ o de la clase ____________.
f) Una definición de clase describe los ____________ y los ____________ de un objeto.
g) Los ocho tipos de datos primitivos de Java son: ____________, ____________, ____________, ____________, ____________, ____________, ____________ y ____________.

**24.6** Diga si es *verdadera* o *falsa* cada uno de las siguientes frases. Si es *falsa*, explique por qué.
a) El método **drawRect** requiere cuatro argumentos que especifiquen dos puntos en el applet, para dibujar un rectángulo.
b) El método **drawLine** requiere cuatro argumentos que especifiquen dos puntos en el applet, para dibujar una línea.
c) El tipo **Double** es un tipo de dato primitivo.
d) El tipo de dato **int** se utiliza para declarar un número de punto flotante.
e) El método **Double.parseDouble** convierte una **String** a un valor primitivo **double**.

**24.7** Escriba las instrucciones Java para llevar a cabo cada una de las siguientes tareas:
a) Despliegue un diálogo que pida al usuario que introduzca un número de punto flotante.
b) Convierta una **String** a un número de punto flotante y almacene el valor convertido en la variable **double edad**. Asuma que la cadena se almacena en **valorCadena**.
c) Dibuje el mensaje **"Este es un programa en Java"** en una línea de un applet (asuma que usted define esta instrucción en el método **paint** del applet) en la posición *(10, 10)*.
d) Dibuje el mensaje **"Este es un programa en Java"** en dos líneas de un applet (asuma que estas instrucciones se definen en el método **paint** del applet) que inicien en la posición *(10, 10)*, y en donde la primera línea termina con **programa**. Haga que las dos líneas comiencen en la misma coordenada *x*.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**24.1** a) //. b) **JOptionPane**. c) Palabras reservadas. d) **main**. e) **System.out.print** y **System.out-.println**. f) Estático.

**24.2** a) Falso. Los comentarios no provocan la ejecución de acción alguna durante la ejecución del programa. Se utilizan para documentar los programas y mejorar su legibilidad.
b) Verdadero.
c) Falso. Java es sensible a mayúsculas y a minúsculas, de modo que las variables son distintas.
d) Falso. El método **Integer.parseInt** convierte una cadena a un valor entero (**int**).

**24.3** a) **`int c, estaEsUnaVariable, q76354, numero;`**
b) **`valor = JOptionPane.showInputDialog( "Introduzca un entero " );`**
c) **`edad = Integer.parseInt( valorCadena );`**
d)
```
if ( numero != 7 )
 JOptionPane.showMessageDialog( null,
    "La variable numero no es igual a 7" );
```
e) **`System.out.println( "Este es un programa en Java" );`**
f) **`System.out.println( "Este es un programa\n en Java" );`**

**24.4** Error: el operador relacional **`=>`** es incorrecto.
Corrección: modifique **`=>`** por **`<=`**.

**24.5** a) **`Graphics`**. b) **`init`**, **`start`** y **`paint`**. c) **`drawLine`** y **`drawRect`**. d) **`extends`** e) **`JApplet`**, **`Applet`**. f) Atributos y comportamientos. g) **`byte`**, **`short`**, **`int`**, **`float`**, **`double`**, **`char`** y **`boolean`**.

**24.6** a) Falso. El método **`drawRect`** requiere cuatro argumentos, dos que especifiquen la esquina superior izquierda del rectángulo y dos que especifiquen el ancho y la altura.
b) Verdadero.
c) Falso. El tipo **`Double`** es una clase dentro del paquete **`java.lang`**. Recuerde que, por lo general, los nombres que comienzan con una letra mayúscula son nombres de clases.
d) Falso. El tipo de dato **`double`** o el tipo de dato **`float`** pueden utilizarse para declarar un número de punto flotante. El tipo de dato **`int`** se utiliza para declarar enteros.
e) Verdadero.

**24.7** a)
```
valor = JOptionPane.showInputDialog(
      "Introduzca un número de punto flotante" );
```
b) **`edad = Double.parseDouble( valorCadena );`**
c) **`g.drawString( "Este es un programa en Java", 10, 10 );`**
d)
```
g.drawString( "Este es un programa", 10, 10 );
g.drawString( "en Java", 10, 25 );
```

## EJERCICIOS

**24.8** Complete los espacios en blanco.
a) Los ____________ se utilizan para documentar un programa y mejorar su legibilidad.
b) Un diálogo de entrada capaz de recibir la entrada del usuario se despliega con el método ____________ de la clase ____________.

**24.9** Escriba una instrucción en Java que lleve a cabo cada una de las siguientes tareas:
a) Despliegue el mensaje **`"Introduzca dos números"`** por medio de la clase **`JOptionPane`**.
b) Asigne el producto de las variables **`b`** y **`c`** a la variable **`a`**.
c) Indique que un programa realiza un cálculo de nómina (es decir, utilice texto que ayude a documentar el programa).

**24.10** ¿Qué se despliega en el diálogo de mensaje cuando se ejecutan cada una de las siguientes instrucciones de Java? Asuma que **`x = 2`** y **`y = 3`**.
a) **`JOptionPane.showMessageDialog( null, "x = " + x );`**
b)
```
JOptionPane.showMessageDialog( null,
   "El valor de x + x es " + ( x + x ) );
```
c) **`JOptionPane.showMessageDialog( null, "x = " );`**
d)
```
JOptionPane.showMessageDialog( null,
   ( x + y ) + " = " + ( y + x ) );
```

**24.11** Escriba una aplicación que solicite al usuario que introduzca dos números, que obtenga dos números del usuario y que imprima la suma, el producto, la diferencia y el cociente de los dos números. Utilice las técnicas mostradas en la figura 24.7.

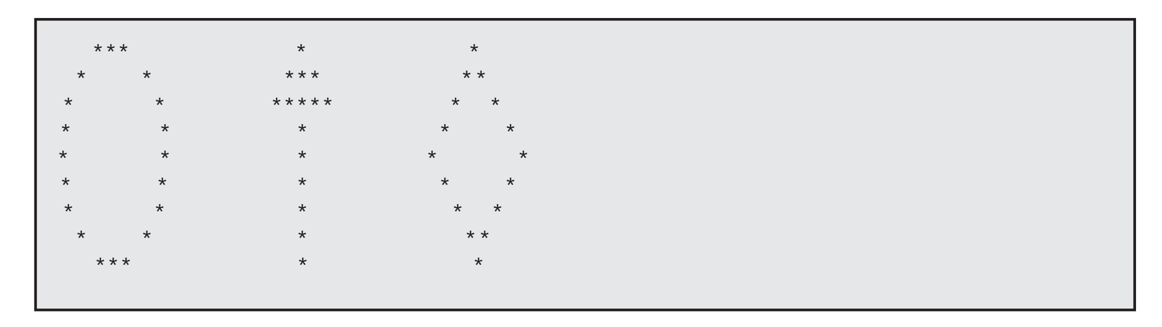
**24.12** Escriba una aplicación que solicite al usuario que introduzca dos enteros, que obtenga los números del usuario y que despliegue el número más grande seguido por las palabras **`"es mayor"`** dentro de un diálogo de mensaje de información. Si los números son iguales, que imprima el mensaje **`"Estos numeros son iguales"`**. Utilice las técnicas mostradas en la figura 24.7.

**24.13** Escriba una aplicación que introduzca tres enteros del usuario y que despliegue la suma, el promedio, el producto, el más pequeño y el más grande de estos números dentro de un diálogo de mensaje de información. Utilice las téc-

nicas para GUI mostradas en la figura 24.7. [*Nota:* El cálculo del promedio en este ejercicio debe ser una representación entera del promedio. Así, si la suma de los valores es 7, el promedio será 2 y no 2.333…]

**24.14** Escriba una aplicación que introduzca el radio de un círculo por parte del usuario y que imprima el diámetro, la circunferencia y el área del círculo. Utilice el valor constante **`3.14159`** para π. Utilice las técnicas de GUI mostradas en la figura 24.7. [*Nota:* Podría utilizar también la constante predefinida **`Math. PI`** para el valor de π. Esta constante es más precisa que el valor **`3.14159`**. La clase **`Math`** está definida dentro del paquete **`java.lang`**, de modo que usted no necesita importarla.] Utilice las siguientes fórmulas ($r$ es el radio): *diámetro* = $2r$, *circunferencia* = $2\pi r$, *área* = $\pi r^2$.

**24.15** Escriba una aplicación que despliegue en la ventana de comando una caja, una elipse, una flecha y un rombo mediante el uso de asteriscos (**`*`**), de la siguiente forma:



```
   ***          *          *
  *   *        ***        ***
 *     *      *****      *   *
 *     *        *       *     *
*       *       *      *       *
 *     *        *       *     *
 *     *        *        *   *
  *   *         *         **
   ***          *          *
```

**24.16** Modifique el programa que creó en el ejercicio 24.15 para desplegar las formas dentro del diálogo **`JOptionPane.PLAIN_MESSAGE`**.

**24.17** Escriba un programa que lea el nombre y el apellido del usuario como dos entradas separadas, y que concatene el nombre y el apellido separados por un espacio. Despliegue el nombre concatenado dentro de un diálogo de mensaje.

# 25

# Más allá de C y C++: Operadores, métodos y arreglos en Java

## Objetivos

- Comprender cómo se utilizan los tipos primitivos y los operadores lógicos en Java.
- Introducir los métodos matemáticos comunes disponibles en la API de Java.
- Crear nuevos métodos.
- Comprender los mecanismos utilizados para pasar información entre métodos.
- Introducir técnicas de simulación, utilizando generación de números aleatorios.
- Comprender los objetos de arreglos en Java.
- Comprender cómo escribir y utilizar métodos que se invocan a sí mismos.




*La forma siempre sigue a la función.*
Louis Henri Sullivan

*E pluribus unum*.
(*Uno compuesto por muchos*.)
Virgilio

*¡Oh! Volvió a llamar ayer, ofreciéndome volver*.
William Shakespeare, *Ricardo II*

*Llámame Ismael*.
Herman Melville, *Moby Dick*

*Cuando me llames así, sonríe.*
Owen Wister

## Plan general



## 25.1 Introducción

En este capítulo presentamos algunas diferencias clave entre Java, C y C++. Comenzamos presentando tipos de datos primitivos y palabras reservadas de Java. Después explicamos los operadores lógicos y métodos, así como los paquetes que comprenden la *Interfaz de programación de aplicaciones* (*API*) de Java.

En el capítulo 5 escribimos un simulador para jugar el juego de craps. En la sección 25.7 retomamos este ejemplo, donde agregamos una *interfaz gráfica de usuario (GUI) y explicamos cómo generar números aleatorios en Java. Finalizamos el capítulo con una explicación sobre arreglos en Java, y cómo mejoran los arreglos en C y C++*.

## 25.2 Tipos de datos primitivos y palabras reservadas

La tabla de la figura 25.1 lista los tipos de datos primitivos en Java. Los tipos primitivos son bloques de construcción para tipos más complicados. Como sus lenguajes predecesores C y C++, Java requiere que todas las variables tengan un tipo antes de que puedan utilizarse en un programa. Por esta razón, Java se conoce como un *lenguaje fuertemente basado en tipos*.

| Tipo | Tamaño en bits | Valores | Estándar |
|---|---|---|---|
| **booleano** | | **verdadero o falso** | |
| **char** | 16 | **'\u000'** a **'\uFFFF'** | (conjunto de caracteres de ISO Unicode) |
| **byte** | 8 | −128 a +127 | |
| **short** | 16 | −32,768 a +32,767 | |
| **int** | 32 | −2,147,483,648 a +2,147,483,647 | |
| **long** | 64 | −9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 | |
| **float** | 32 | −3.40292347E+38 a +3.40292347E+38 | (punto flotante IEEE 754) |
| **double** | 64 | −1.79769313486231570E+308 a +1.79769313486231570E+308 | (punto flotante IEEE 754) |

**Figura 25.1** Los tipos de datos primitivos en Java.

A diferencia de C y de C++, los tipos primitivos en Java son portables a través de todas las plataformas de cómputo que soportan Java. Ésta y muchas otras características de portabilidad de Java permiten a los programadores escribir programas una vez, sin conocer la plataforma de cómputo que ejecutará el programa. En ocasiones, a esto se le conoce como "WORA" (Write Once Run Anywhere; Escríbelo una vez, ejecútalo en donde sea).

En programas en C y C++, los programadores con frecuencia tenían que escribir versiones separadas de sus programas para que los soportaran diferentes plataformas, ya que no se garantizaba que los tipos de datos primitivos fueran idénticos de computadora a computadora. Por ejemplo, un valor **`int`** en una máquina podía representarse con 16 bits (2 bytes) de memoria, y en otra computadora con 32 bits (4 bytes) de memoria. En Java, los valores **`int`** siempre son de 32 bits (4 bytes).

**Tip de portabilidad 25.1**

*Todos los tipos de datos primitivos en Java son portables, a través de todas las plataformas que soportan Java.*

Cada tipo de dato de la tabla se lista con su tamaño en bits (hay 8 bits por un byte), y su rango de valores. Los diseñadores de Java quieren un máximo de portabilidad, por lo que eligieron utilizar estándares internacionalmente reconocidos para formatos de caracteres (Unicode) y para números de punto flotante (IEEE 754).

Siempre que en una clase se declaran instancias de variables de tipos de datos primitivos, se asignan valores predeterminados, a menos que el programador especifique lo contrario. A las variables de tipo **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** y **`double`** se les da el valor de **`0`** de manera predeterminada. A las variables de tipo **`boolean`** se les da de manera predeterminada el valor de **`false`**.

Cada una de las palabras **`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** y **`double`**, son palabras reservadas de Java. Estas palabras están reservadas para el lenguaje, para implementar diversas características, como los tipos de datos primitivos. Las palabras reservadas no pueden utilizarse como identificadores, tal como nombres de variables. En la figura 25.2 aparece una lista completa de las palabras reservadas de Java.



**Error común de programación 25.1**

*Utilizar una palabra reservada como identificador, es un error de sintaxis.*

## 25.3 Operadores lógicos

Java proporciona operadores lógicos que pueden utilizarse para formar condiciones complejas que controlen estructuras mediante la combinación de condiciones simples. Los operadores lógicos son **&&** (*AND lógico*), **&** (*AND lógico booleano*), **`||`** (*OR lógico*), **`|`** (*OR lógico booleano incluyente*), **^** (*OR lógico booleano excluyente*), y **`!`** (*NOT lógico*, también llamado *negación lógica*). Más adelante consideraremos ejemplos de cada uno de ellos.

**Palabras reservadas de Java**

| | | | | |
|---|---|---|---|---|
| **`abstract`** | **`boolean`** | **`break`** | **`byte`** | **`case`** |
| **`catch`** | **`char`** | **`class`** | **`continue`** | **`default`** |
| **`do`** | **`double`** | **`else`** | **`extends`** | **`false`** |
| **`final`** | **`finally`** | **`float`** | **`for`** | **`if`** |
| **`implements`** | **`import`** | **`instanceof`** | **`int`** | **`interface`** |
| **`long`** | **`native`** | **`new`** | **`null`** | **`package`** |
| **`private`** | **`protected`** | **`public`** | **`return`** | **`short`** |
| **`static`** | **`super`** | **`switch`** | **`synchronized`** | **`this`** |
| **`throw`** | **`throws`** | **`transient`** | **`true`** | **`try`** |
| **`void`** | **`volatile`** | **`while`** | | |
| *Palabras que son reservadas, pero que no se utilizan en Java* | | | | |
| **`const`** | **`goto`** | | | |

**Figura 25.2** Palabras reservadas de Java.

Suponga que deseamos garantizar en algún punto de un programa que dos condiciones son **true**, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador lógico **&&** de la siguiente manera:

```
if( genero == 1 && edad >= 65 )
   ++MujeresTerceraEdad;
```

Esta instrucción **if** contiene dos condiciones simples. La condición **genero == 1** puede evaluarse, por ejemplo, para determinar si una persona es mujer. La condición **edad >= 65** se evalúa para determinar si una persona es un ciudadano de la tercera edad. Las dos condiciones simples se evalúan primero, ya que las precedencias de **==** y de **>=** son más altas que la precedencia de **&&.** Después, la instrucción **if** considera la condición combinada

```
genero == 1 && edad >= 65
```

Esta condición es **true** *si y sólo si* ambas condiciones simples son **true**. Por último, si esta condición combinada es **true**, la cuenta de **MujeresTerceraEdad** se incrementa en **1**. Si una o ambas condiciones son **false**, el programa evita el incremento y continúa con la instrucción siguiente a la estructura **if**. La condición combinada anterior puede hacerse más legible, agregando paréntesis redundantes:

```
( genero == 1 ) && ( edad >= 65 )
```

La tabla de la figura 25.3 resume el operador **&&**. La tabla muestra las cuatro posibles combinaciones de valores **false** y **true** para la *expresion1* y la *expresion2*. A tales tablas con frecuencia se les conoce como *tablas de verdad*. Java da como resultado **false** o **true** para todas las expresiones que incluyen operadores de relación, de igualdad y/o operadores lógicos.

Ahora consideremos el operador **||** (OR lógico). Suponga que deseamos garantizar que una *o* ambas condiciones sean **true**, antes de elegir una cierta ruta de ejecución. En este caso, utilizamos el operador **||** como en el siguiente segmento de programa:

```
if( promedioSemestre >= 90 || examenFinal >= 90 )
   System.out.println( "La calificacion del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición **promedioSemestre >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un buen desempeño a lo largo del semestre. La condición **examenFinal >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un desempeño sobresaliente en el examen final. La instrucción **if** después considera la condición combinada

```
promedioSemestre >= 90 || examenFinal >= 90
```

y otorga al estudiante una “A”, si una o ambas condiciones simples son **true**. Observe que el mensaje **“La calificacion del estudiante es A”**, *no* se imprime sólo cuando ambas condiciones simples son **false**. La figura 25.4 es una tabla de verdad para el operador lógico OR (**||**).

El operador **&&** tiene una precedencia más alta que el operador **||**. Ambos operadores asocian de izquierda a derecha. Una expresión que contiene los operadores **&&** o **||** se evalúa sólo hasta que se conoce su veracidad o su falsedad. Por lo tanto, la evaluación de la expresión **genero == 1 && edad >= 65** se detendrá inmediatamente si **genero** no es igual que **1** (es decir, la expresión completa es **false**), y continuará si **genero** es igual que **1** (es decir, la expresión completa podría seguir siendo **true**, si la condición **edad >= 65** es

| expresion1 | expresion2 | expresion1 && expresion2 |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

**Figura 25.3** Tabla de verdad para el operador **&&** (Y lógico).

| expresion1 | expresion2 | expresion1 \|\| expresion2 |
|---|---|---|
| `false` | `false` | `false` |
| `false` | `true` | `true` |
| `true` | `false` | `true` |
| `true` | `true` | `true` |

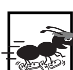
**Figura 25.4** Tabla de verdad para el operador **||** (OR lógico).

**true**). Esta característica de desempeño para la evaluación de expresiones lógicas AND y OR se conoce como *evaluación en cortocircuito*.



**Error común de programación 25.2**

*En expresiones que utilizan el operador* ***&&****, es posible que una condición (a la que llamaremos condición dependiente) requiera de otra condición para ser* ***true****, de tal modo que ésta tenga sentido al evaluar la condición dependiente. En este caso, la condición dependiente debe colocarse después de la otra condición, o es posible que ocurra un error.*



**Tip de rendimiento 25.1**

*En expresiones que utilizan el operador* ***&&****, si las condiciones separadas son independientes una de la otra, haga que la condición que más probablemente sea falsa, se encuentre más a la izquierda. En expresiones que utilizan el operador* ***||****, haga que la condición que más probablemente sea verdadera, se encuentre más a la izquierda. Esto puede reducir el tiempo de ejecución de un programa.*

Los operadores *AND lógico booleano* (**&**) y *OR lógico booleano incluyente* (**|**) funcionan de manera idéntica a los operadores lógicos AND y OR, con una excepción: los operadores lógicos booleanos siempre evalúan sus dos operandos (es decir, no hay una evaluación en cortocircuito). Por lo tanto, la expresión

```
genero == 1 & edad >= 65
```

evalúa **edad >= 65** independientemente de si **genero** es igual que **1**. Esto es útil si el operando derecho del operador lógico booleano AND, o el operador lógico booleano incluyente **OR** tiene un *efecto colateral* necesario; una modificación al valor de una variable. Por ejemplo, la expresión

```
cumpleanios == true | ++edad >= 65
```

garantiza que la condición **++edad >= 65** será evaluada. Entonces, la variable **edad** se incrementará en la expresión anterior, independientemente de si la expresión completa es **true** o **false**.



**Buena práctica de programación 25.1**

*Por claridad, evite expresiones con efectos colaterales en las condiciones. Los efectos colaterales pueden parecer convenientes, pero con frecuencia representan más problemas que ventajas.*

Una condición que contiene el operador *OR lógico booleano excluyente (^)* es **true**, *si y sólo si uno de sus operandos resulta en un valor* ***true*** *y uno resulta en un valor* ***false***. Si ambos operandos son **true**, o ambos son **false**, el resultado de la condición completa es **false**. La figura 25.5 es una tabla de verdad para

| expresion1 | expresion2 | expresion1 ^ expresion2 |
|---|---|---|
| `false` | `false` | `false` |
| `false` | `true` | `true` |
| `true` | `false` | `true` |
| `true` | `true` | `false` |

**Figura 25.5** Tabla de verdad para el operador lógico booleano excluyente OR (^).

el operador lógico booleano excluyente OR (**^**). Este operador también garantiza la evaluación de sus dos operandos (es decir, no existe una evaluación de cortocircuito).

Java proporciona el operador **!** (negación lógica) para permitir al programador "revertir" el significado de una condición. A diferencia de los operadores lógicos **&&**, **&**, **||**, **|** y **^**, los cuales combinan dos condiciones (operadores binarios), el operador de negación lógica tiene solamente una condición como operando (operador unario). El operador de negación lógica se coloca antes de una condición para elegir una ruta de ejecución, si la condición original (sin el operador de negación lógica) es **false**, como en el siguiente segmento de programa:

```
if ( ! ( calificacion == valorCentinela ) )
   System.out.println( "La siguiente calificacion es " + calificacion );
```

Los paréntesis alrededor de la condición **calificacion == valorCentinela** son necesarios, ya que el operador de negación lógica tiene una precedencia más alta que el operador de igualdad. La figura 25.6 es una tabla de verdad para el operador de negación lógica.

En la mayoría de los casos, el programador puede evitar el uso de la negación lógica, expresando la condición de manera diferente con un operador de igualdad o de relación adecuado. Por ejemplo, la instrucción anterior puede escribirse de la siguiente manera:

```
if( calificacion != valorCentinela )
   System.out.printl( "La siguiente calificacion es " + calificacion );
```

Esta flexibilidad puede ayudar al programador a expresar una condición de una manera más conveniente. La aplicación de la figura 25.7 muestra todos los operadores lógicos y booleanos, produciendo sus tablas de verdad. El programa utiliza la concatenación de cadenas para crear la cadena que se despliega en un **JTextArea**.

En la salida de la figura 25.7, las cadenas "verdadero" y "falso" indican **false** y **true** para los operandos de cada condición. El resultado de la condición aparece como **true** o **false**. Observe que siempre que agrega un valor **boolean** a una **String**, Java agrega la cadena "false" o "true", basándose en el valor booleano.

| expresion | !expresion |
|---|---|
| **false** | **true** |
| **true** | **false** |

**Figura 25.6** Tabla de verdad para el operador !( NOT lógico).

```
// Figura 25.7: OperadoresLogicos.java
// Demostración de los operadores lógicos
import javax.swing.*;

public class OperadoresLogicos {
   public static void main( String args[] )
   {
      JTextArea areaSalida = new JTextArea( 17, 20 );
      JScrollPane desplaza = new JScrollPane( outputArea );
      String salida = "";

      salida += " AND Logico (&&)" +
                 "\nfalso && falso: " + ( false && false ) +
                 "\nfalso && verdadero: " + ( false && true ) +
                 "\nverdadero && falso: " + ( true && false ) +
                 "\nverdadero && verdadero: " + ( true && true );
```

**Figura 25.7** Demostración de los operadores lógicos. (Parte 1 de 2.)

```

        salida += “\n\n OR Logico (||)” +
                   “\nfalso || falso: “ + ( false || false ) +
                   “\nfalso || verdadero: “ + ( false || true ) +
                   “\nverdadero || falso: “ + ( true || false ) +
                   “\nverdadero || verdadero: “ + ( true || true );

        salida += “\n\nAND Logico Booleano (&)” +
                   “\nfalso & falso: “ + ( false & false ) +
                   “\nfalso & verdadero: “ + ( false & true ) +
                   “\nverdadero & falso: “ + ( true & false ) +
                   “\nverdadero & verdadero: “ + ( true & true );

        salida += “\n\n OR Logico Booleano Incluyente (|)” +
                   “\nfalso | falso: “ + ( false | false ) +
                   “\nfalso | verdadero: “ + ( false | true ) +
                   “\nverdadero | falso: “ + ( true | false ) +
                   “\nverdadero | verdadero: “ + ( true | true );

        salida += “\n\nOR Logico Booleano Excluyente (^)” +
                   “\nfalso ^ falso: “ + ( false ^ false ) +
                   “\nfalso ^ verdadero: “ + ( false ^ true ) +
                   “\nverdadero ^ falso: “ + ( true ^ false ) +
                   “\nverdadero ^ verdadero: “ + ( true ^ true );

        salida += “\n\nNOT Logico (!)” +
                   “\n!falso: “ + ( !false ) +
                   “\n!verdadero: “ + ( !true );

        outputArea.setText( salida );
        JOptionPane.showMessageDialog( null, scroller,
           “Tablas de verdad”, JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    } // fin de main
} // fin de la clase OperadoresLogicos
```

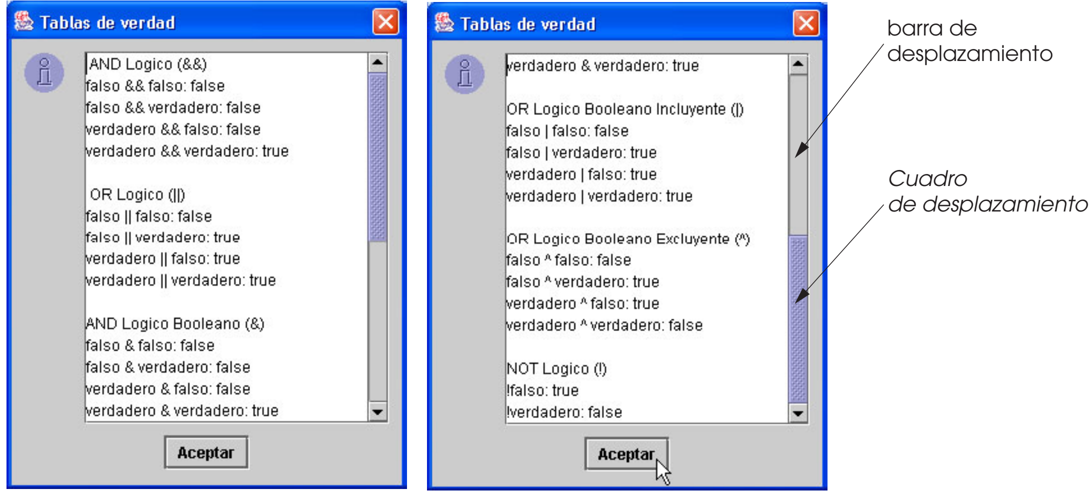



**Figura 25.7** Demostración de los operadores lógicos. (Parte 2 de 2.)

La línea 8 del método main

```
JTextArea areaSalida = new JTextArea( 17, 20 );
```

crea un **JTextArea** con 17 filas y 20 columnas. La línea 9

```
JScrollPane desplaza = new JScrollPane( areaSalida );
```

declara la referencia **desplaza** de **JScrollPane** y lo inicializa con un nuevo objeto **JScrollPane**. La clase **JScrollPane** (del paquete **javax.swing**) proporciona un componente GUI con funcionalidad de desplazamiento.

Cuando ejecute esta aplicación, observe la *barra de desplazamiento* del lado derecho de **JTextArea**. Puede hacer clic en las *flechas* superior o inferior de la barra de desplazamiento para desplazarse hacia arriba o hacia abajo a lo largo del texto del **JTextArea**, una línea a la vez. También puede arrastrar el *cuadro de desplazamiento* (también llamado el *pulgar*) hacia arriba o hacia abajo, para desplazarse rápidamente por el texto. Un objeto **JScrollPane** se inicializa con el componente GUI para el que proporcionará la funcionalidad de desplazamiento (en este caso, **areaSalida**). Esto adjunta el componente GUI al **JScrollPane**.

Las líneas 12 a 44 construyen la cadena **salida** que se desplegará en el **areaSalida**. La línea 46 utiliza el método **setText** para remplazar el texto de **areaSalida** con el de la cadena **salida**. Las líneas 47 y 48 despliegan un diálogo de mensaje. El segundo argumento, **desplaza**, indica que el **desplaza** y el **areaSalida** adjunto a él deben desplegarse como el mensaje del diálogo de mensaje.

## 25.4 Definiciones de métodos

Todos los programas que hemos presentado consisten en una definición de clase que al menos contiene una definición de métodos, llamados métodos API de Java, para realizar sus tareas. Ahora consideraremos cómo es que los programadores escriben sus propios métodos personalizados.

Considere un applet (figura 25.8) que utiliza un método **cuadrado** (invocado desde el método **init** del applet) para calcular los cuadrados de enteros en el rango de 1 a 10.

Cuando el applet comienza su ejecución, se llama primero a su método **init**. La línea 9 declara la referencia **salida** de **String**, y la inicializa con la cadena vacía. Esta **String** contendrá los resultados de elevar al cuadrado los valores de 1 a 10. La línea 11 declara la referencia **areaSalida** de **JTextArea**, y la

```
1  // Figura 25.8: CuadradoEnt.java
2  // Método cuadrado definido por el programador
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class CuadradoEnt extends JApplet {
7     public void init()
8     {
9        String salida = “”;
10
11       JTextArea areaSalida = new JTextArea( 10, 20 );
12
13       // obtiene el área de visualización del componente GUI del applet
14       Container c = getContentPane();
15
16       // adjunta el areaSalida al Contenedor c
17       c.add( areaSalida );
18
19       int resultado;
20
```

**Figura 25.8** Uso del método **cuadrado** definido por el programador. (Parte 1 de 2.)

```
21          for ( int x = 1; x <= 10; x++ ) {
22             resultado = cuadrado( x );
23             salida += "El cuadrado de " + x +
24                        " es " + resultado + "\n";
25          } // fin de for
26
27          areaSalida.setText( salida );
28       } // fin del método init
29
30       // definición del método cuadrado
31       public int cuadrado( int y )
32       {
33          return y * y;
34       } // fin del método cuadrado
35    } // fin de la clase CuadradoEnt
```

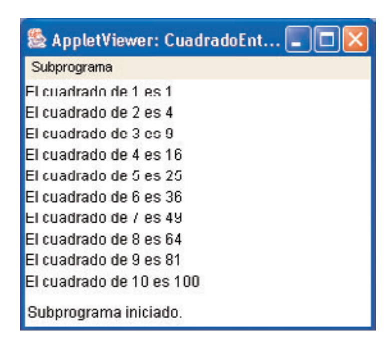



**Figura 25.8** Uso del método **`cuadrado`** definido por el programador. (Parte 2 de 2.)

inicializa con un nuevo objeto **`JTextArea`** de 10 filas y 20 columnas. La cadena **`salida`** se desplegará en **`areaSalida`**.

Este programa es el primero en el que desplegamos un componente GUI en un applet. El área de la pantalla en la que se despliega un **`JApplet`** tiene un *panel de contenido* en el que los componentes GUI deben adjuntarse, de tal modo que puedan desplegarse en tiempo de ejecución. El panel de contenido es un objeto de la clase *Container* del *paquete* ***`java.awt`***. Esta clase se importó en la línea 3 para utilizarla en el applet. La línea 14

```
Container c = getContentPane();
```

declara la referencia **`c`** de **`Container`**, y la asigna al resultado de una llamada al método ***`getContentPane`***; uno de los muchos métodos que nuestra clase **`CuadradoEnt`** hereda de la clase **`JApplet`**. El método **`getContentPane`** devuelve una referencia al panel de contenido del applet, la cual puede utilizarse para adjuntar componentes GUI, tales como un **`JTextArea`**, a la interfaz de usuario de la applet.

La línea 17

```
c.add( areaSalida );
```

coloca en el applet el objeto componente de la GUI, **`JTextArea`**, al que hace referencia **`areaSalida`**, para que pueda desplegarse cuando el applet se ejecuta. El método **`add`** de **`Container`** adjunta un componente GUI al contenedor. Por el momento, sólo podemos adjuntar un componente GUI al panel de contenido del applet, y ese componente GUI automáticamente ocupará toda el área de dibujo del applet en la pantalla (como definieron el **`width`** y la **`height`** del applet en pixeles, en el documento HTML del applet). Más adelante explicaremos cómo distribuir varios componentes GUI en un applet.

La línea 19 declara la variable **`int`**, **`resultado`**, en la que se almacena el resultado de calcular cada cuadrado. Las líneas 21 a 25 corresponden a una estructura **`for`**, en la que cada iteración del ciclo calcula el

**cuadrado** del valor actual de la variable de control **x**, almacena el valor en **resultado** y concatena el **resultado** al final de **salida**.

El método **cuadrado** se *invoca* o se *llama* en la línea 22, por medio de la instrucción

```
resultado = cuadrado( x );
```

Cuando el control del programa alcanza esta instrucción, el método **cuadrado** (definido en la línea 31) es invocado. De hecho, los **()** representan el *operador de llamada a métodos*, el cual tiene una precedencia alta. En este punto, el programa hace automáticamente una copia del valor de **x** (el *argumento* de la llamada al método), y el control del programa se transfiere a la primera línea del método **cuadrado**. El método **cuadrado** recibe la copia del valor de **x** en el *parámetro* **y**. Después, **cuadrado** calcula **y * y**. El resultado se pasa de regreso hacia el punto en **init** donde se invocó a **cuadrado**. El valor devuelto se asigna entonces a la variable **resultado**. Las líneas 23 y 24

```
salida += "El cuadrado de " + x +
          " es " + resultado + "\n";
```

concatenan **"El cuadrado de "**, el valor de **x**, **"es "**, el valor de **resultado**, y un carácter de nueva línea al final de **salida**. Este proceso se repite diez veces, por medio de la estructura de repetición **for**. La línea 27

```
areaSalida.setText( salida );
```

utiliza el método **setText** para establecer el texto de **areaSalida** a la **String salida**. Observe que las referencias **salida**, **areaSalida**, **c** y la variable **resultado** se declaran como variables locales en **init**, ya que sólo se utilizan en **init**. Las variables deben declararse como variables de instancia, sólo si es necesario utilizarlas en más de un método de la clase, o si sus valores deben guardarse entre llamadas a los métodos de la clase.

La definición del método **cuadrado** (línea 31) muestra que **cuadrado** espera un parámetro entero **y**; éste será el nombre utilizado para manipular el valor pasado a **cuadrado** en el cuerpo del método. La palabra reservada **init** que precede al nombre del método indica que **cuadrado** devuelve un resultado entero. La *instrucción* ***return*** de **cuadrado** pasa el resultado del cálculo **y * y** de regreso al método que hizo la llamada. Observe que la definición completa del método se encuentra entre las llaves de la clase **CuadradoEnt**. Todos los métodos deben declararse dentro de una definición de clase.



**Buena práctica de programación 25.2**

*Coloque una línea en blanco entre las definiciones de métodos para separarlos y para mejorar la legibilidad del programa.*



**Error común de programación 25.3**

*Definir un método fuera de las llaves correspondientes a la definición de una clase, es un error de sintaxis.*

El formato para la definición de un método es

*tipo del valor de retorno nombre del método (lista de parámetros)*
{
*declaraciones e instrucciones*
}

El *nombre del método* es cualquier identificador válido. El *tipo del valor de retorno* es el tipo de dato del resultado devuelto por el método a quien hizo la llamada. El tipo del valor de retorno **void** indica que un método no devuelve valor alguno. Los métodos pueden devolver, cuando mucho, un valor.



**Error común de programación 25.4**

*Omitir el tipo del valor de retorno en la definición de un método, es un error de sintaxis.*



**Error común de programación 25.5**

*Olvidar devolver un valor por parte de un método que se supone debe hacerlo, es un error de sintaxis. Si se especifica un tipo de valor de retorno diferente de* ***void****, el método debe contener una instrucción* ***return****.*



### Error común de programación 25.6

*Devolver un valor desde un método, cuyo tipo de retorno se declaró como **`void`**, es un error de sintaxis.*

La *lista de parámetros* es una lista separada por comas que contiene las declaraciones de los parámetros recibidos por el método cuando es llamado. En la llamada al método debe haber un argumento para cada parámetro en la definición del método. Los argumentos también deben ser compatibles con el tipo del parámetro. Por ejemplo, un tipo de parámetro **`double`** podría recibir valores de **`7.35`**, **`22`**, **`o -0.3546`**, pero no **`"hola"`** (ya que una variable **`double`** no puede contener un **`String`**). Si un método no recibe valores, la *lista de parámetros* está vacía (es decir, al nombre del método le sigue un conjunto de paréntesis vacío). Un tipo debe listarse explícitamente para cada parámetro de la lista de un método, u ocurrirá un error de sintaxis.



### Error común de programación 25.7

*Declarar parámetros del mismo tipo en un método, como **`float x, y`**, en lugar de **`float x, float y`**, es un error de sintaxis, ya que se necesitan tipos para cada parámetro de la lista de parámetros.*



### Error común de programación 25.8

*Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de una definición de método, es un error de sintaxis.*



### Error común de programación 25.9

*Redefinir un parámetro de un método como una variable local del método, es un error de sintaxis.*



### Error común de programación 25.10

*Pasar un método a un argumento que no es compatible con el tipo correspondiente al parámetro, es un error de sintaxis.*



### Buena práctica de programación 25.3

*Aunque no es incorrecto hacerlo, en la definición de un método no utilice los mismos nombres para los argumentos pasados a él y para los parámetros correspondientes. Esto ayuda a evitar la ambigüedad.*

Las *declaraciones* e *instrucciones* entre llaves forman el *cuerpo del método*. Al cuerpo del método también se le conoce como *bloque*. Un bloque es una instrucción compuesta que incluye declaraciones. Las variables pueden declararse en cualquier bloque, y los bloques pueden estar anidados. Un método no puede definirse dentro de otro método.



### Error común de programación 25.11

*Definir un método dentro de otro, es un error de sintaxis.*



### Buena práctica de programación 25.4

*Elegir nombres descriptivos para los métodos y para los parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.*



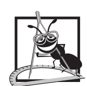
### Observación de ingeniería de software 25.1

*Por lo general, un método no debe sobrepasar una página. Mejor aún, un método generalmente debe abarcar no más de media página. Independientemente de cuán largo sea un método, debe realizar bien una tarea. Los métodos pequeños promueven la reutilización de software.*



### Tip para prevenir errores 25.1

*Los métodos pequeños son más fáciles de probar, depurar y comprender, que aquellos que son grandes.*



### Observación de ingeniería de software 25.2

*Los programas deben escribirse como colecciones de métodos pequeños. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.*

### Observación de ingeniería de software 25.3

*Es posible que un método que requiere un gran número de parámetros esté realizando demasiadas tareas. Considere dividir el método en métodos más pequeños que realicen tareas separadas. Si es posible, el encabezado del método debe caber en una línea.*



### Observación de ingeniería de software 25.4

*El encabezado de un método y las llamadas a él deben coincidir en número, tipo y orden de parámetros y argumentos.*

Existen tres formas para devolver el control al punto en el que se invocó a un método. Si el método no devuelve un resultado, el control se devuelve cuando se alcanza la llave derecha de terminación del método, o ejecutando la instrucción

```
return;
```

Si el método devuelve un resultado, la instrucción

```
return expresión;
```

devuelve el valor de *expresión* a quien hizo la llamada. Cuando se ejecuta una instrucción **return**, el control vuelve inmediatamente al punto en el que se invocó al método.

Observe que el ejemplo de la figura 25.8 en realidad contiene dos definiciones de métodos; **init** (línea 7) y **cuadrado** (línea 31). Recuerde que el método **init** es llamado automáticamente para inicializar el applet. En este ejemplo, el método **init** invoca de manera reiterada al método **cuadrado** para que realice un cálculo, después despliega los resultados en el **JTextArea** que está adjunto al panel de contenido del applet.

Observe la sintaxis utilizada para invocar al método **cuadrado**; sólo utilizamos el nombre del método, seguido por los argumentos de éste entre paréntesis. Por medio de esta sintaxis, a los métodos en una definición de clase se les permite invocar a todos los métodos restantes de la misma definición de clase (existe una excepción, la cual explicaremos en el capítulo 26). Los métodos en la misma definición de clase son los métodos definidos en esa clase y los métodos heredados (los métodos de la clase que la clase actual **extiende (extends)**; en el último ejemplo, **JApplet**). Ahora hemos visto tres formas para llamar a un método; por el nombre mismo del método (como mostramos con **cuadrado( x )**, en este ejemplo), por medio de una referencia a un objeto seguido por el operador punto (**.**) y por el nombre del método [como en **g.drawline( x1, y1, x2, y2 )**], y por medio del nombre de una clase seguido por el nombre del método [como en **Integer.parseInt( stringToConvert )**]. La última sintaxis es sólo para métodos **static** de una clase (los cuales explicaremos con detalle en el capítulo 26).

## 25.5 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases predefinidas que están agrupadas en directorios del disco, en categorías de clases relacionadas llamadas paquetes. Juntos, estos paquetes se conocen como la interfaz de programación de aplicaciones de Java (API de Java).

| Tipo | Promociones permitidas |
|---|---|
| **double** | Ninguna |
| **float** | **double** |
| **long** | **float** o **double** |
| **int** | **long**, **float** o **double** |
| **char** | **int**, **long**, **float** o **double** |
| **short** | **int**, **long**, **float** o **double** |
| **byte** | **short**, **int**, **long**, **float** o **double** |
| **boolean** | Ninguna (en Java, los valores **booleanos** no se consideran como números) |

**Figura 25.9** Promociones permitidas para tipos de datos primitivos.

A lo largo del texto, utilizamos las instrucciones **`import`** para especificar la ubicación de las clases requeridas para compilar un programa en Java. Por ejemplo, para indicarle al compilador que cargue la clase **`JApplet`** del paquete **`javax.swing`**, se utiliza la instrucción

```
import javax.swing.JApplet;
```

Una de las grandes fortalezas de Java es el gran número de clases en los paquetes de la API de Java, las cuales pueden reutilizar los programadores, en lugar de "reinventar la rueda". En este libro practicamos con muchas de estas clases. La figura 25.10 lista alfabéticamente los paquetes de la API de Java, y proporciona una breve descripción de cada uno. Es posible descargar otros paquetes disponibles, desde **`http://java.sun.com`**. Observe que aún no hemos explicado la mayoría de estos paquetes. Esta tabla se la proporcionamos para darle una idea de la variedad de componentes reutilizables que se encuentran disponibles en la API de Java. Cuando se aprende Java, uno debe invertir tiempo en leer las descripciones de los paquetes y las clases en la documentación de la API de Java.

| Paquete | Descripción |
|---|---|
| **`java.applet`** | *The Java Applet Package*.<br>Este paquete contiene la clase **`Applet`** y diversas interfaces que permiten la creación de applets, la interacción de applets con el navegador y con los clips de reproducción de audio. En Java 2, la clase **`javax.swing.JApplet`** se utiliza para definir un applet que utiliza los *Swing GUI components*. |
| **`java.awt`** | *The Java Abstract Windowing Toolkit Package*.<br>Este paquete contiene las clases e interfaces requeridas para crear y manipular interfaces gráficas de usuario en Java 1.0 y 1.1. En Java 2, estas clases pueden utilizarse, pero con frecuencia, en su lugar se utilizan los *Swing GUI components* de los paquetes de **`javax.swing`**. |
| **`java.awt.color`** | *The Java Color Space Package.*<br>Este paquete contiene clases que soportan espacios de color. |
| **`java.awt.datatransfer`** | *The Java Data Transfer Package*.<br>Este paquete contiene clases e interfaces que permiten la transferencia de datos entre un programa en Java y el portapapeles de la computadora (un área de almacenamiento temporal para datos). |
| **`java.awt.dnd`** | *The Java Drag-and-Drop Package*.<br>Este paquete contiene clases e interfaces que proporcionan capacidades para arrastrar y soltar programas. |
| **`java.awt.event`** | *The Java Abstract Windowing Toolkit Event Package*.<br>Este paquete contiene clases e interfaces que permiten el manejo de eventos para componentes GUI en los paquetes **`java.awt`** y **`javax.swing`**. |
| **`java.awt.font`** | *The Java Font Manipulation Package*.<br>Este paquete contiene clases e interfaces para manipular diferentes fuentes. |
| **`java.awt.geom`** | *The Java Two-Dimensional Objects Package*.<br>Este paquete contiene clases para manipular objetos que representan gráficos bidimensionales. |
| **`java.awt.im`** | *The Java Input Method Framework Package*.<br>Este paquete contiene clases y una interfaz que soporta entrada en los lenguajes japonés, chino y coreano en un programa en Java. |
| **`java.awt.image`**<br>**`java.awt.image.renderable`** | *The Java Image Packages*.<br>Estos paquetes contienen clases e interfaces que permiten el ordenamiento y la manipulación de imágenes en un programa. |

**Figura 25.10** Paquetes de la API de Java. (Parte 1 de 4.)

| Paquete | Descripción |
|---|---|
| `java.awt.print` | *The Java Printing Package.*<br>Este paquete contiene clases e interfaces que soportan la impresión desde programas en Java. |
| `java.beans`<br>`java.beans.beancontext` | *The Java Beans Packages.*<br>Estos paquetes contienen clases e interfaces que permiten al programador crear componentes reutilizables de software. |
| `java.io` | *The Java Input/Output Package.*<br>Este paquete contiene clases que permiten a los programas introducir y desplegar datos. |
| `java.lang` | *The Java Language Package.*<br>Este paquete contiene clases e interfaces requeridas por muchos programas en Java (muchos de cuales explicamos a lo largo del texto), y el compilador lo importa automáticamente hacia todos los programas. |
| `java.lang.ref` | *The Reference Objects Package.*<br>Este paquete contiene clases que permiten la interacción entre un programa en Java y un recolector de basura. |
| `java.lang.reflect` | *The Java Core Reflection Package.*<br>Este paquete contiene clases e interfaces que permiten a un programa descubrir de manera dinámica las variables y los métodos accesibles de una clase durante la ejecución de un programa. |
| `java.math` | *The Java Arbitrary Precision Math Package.*<br>Este paquete contiene clases para realizar aritmética con una precisión arbitraria. |
| `java.net` | *The Java Networking Package.*<br>Este paquete contiene clases que permiten a los programas comunicarse a través de redes. |
| `java.rmi`<br>`java.rmi.activation`<br>`java.rmi.dgc`<br>`java.rmi.registry`<br>`java.rmi.server` | *The Java Remote Method Invocation Packages.*<br>Estos paquetes contienen clases e interfaces que permiten al programador crear programas distribuidos en Java. Al utilizar un método de invocación remoto, un programa puede llamar a un método de un programa separado en la misma computadora, o en una computadora en cualquier parte por medio de Internet. |
| `java.security`<br>`java.security.acl`<br>`java.security.cert`<br>`java.security.interfaces`<br>`java.security.spec` | *The Java Security Packages.*<br>Estos paquetes contienen clases e interfaces que permiten a un programa en Java encriptar los datos y controlar los privilegios de acceso proporcionados a un programa en Java para efectos de seguridad. |
| `java.sql` | *The Java Database Connectivity Package.*<br>Este paquete contiene clases e interfaces que permiten a un programa en Java interactuar con una base de datos. |
| `java.text` | *The Java Text Package.*<br>Este paquete contiene clases e interfaces que permiten a un programa en Java manipular números, fechas, caracteres y cadenas. Este paquete también proporciona muchas de las capacidades de internacionalización de Java para personalizar aplicaciones locales (o de una región geográfica en particular). |

**Figura 25.10** Paquetes de la API de Java. (Parte 2 de 4.)

| Paquete | Descripción |
|---|---|
| **java.util** | *The Java Utilities Package.*<br>Este paquete contiene clases de utilidad e interfaces como: manipulaciones de fecha y hora, capacidades para procesamiento de números aleatorios (**Random**), almacenamiento y procesamiento de grandes cantidades de datos, romper cadenas en piezas pequeñas llamadas tokens (**StringTokenizer**), y otras capacidades. |
| **java.util.jar**<br>**java.util.zip** | *The Java Utilities JAR and ZIP Packages.*<br>Estos paquetes contienen clases de utilidad e interfaces que permiten a un programa en Java combinar archivos de Java **.class** y otros archivos de recursos (como imágenes y audio) en archivos comprimidos llamados *archivos Java archive* (*JAR*) o *archivos ZIP*. |
| **javax.accessibility** | *The Java Accesibility Package.*<br>Este paquete contiene clases e interfaces que permiten a un programa en Java soportar tecnologías para gente con discapacidades; algunos ejemplos son los lectores de pantalla y los magnificadores de pantalla. |
| **java.swing** | *The Java Swing GUI Components Package.*<br>Este paquete contiene clases e interfaces para los componentes Swing GUI de Java que proporcionan soporte para GUIs portables. |
| **javax.swing.border** | *The Java Swing Borders Package.*<br>Este paquete contiene clases y una interfaz para dibujar límites alrededor de áreas en una GUI. |
| **javax.swing.colorchooser** | *The Java Swing Color Chooser Package.*<br>Este paquete contiene clases e interfaces para el diálogo predefinido **JColorChooser** para elegir colores. |
| **javax.swing.event** | *The Java Swing Event Package.*<br>Este paquete contiene clases e interfaces que permiten la manipulación de eventos para componentes GUI en el paquete **javax.swing**. |
| **javax.swing.filechooser** | *The Java Swing File Chooser Package.*<br>Este paquete contiene clases e interfaces para el diálogo predefinido **JFileChooser** para localizar archivos en disco. |
| **javax.swing.plaf**<br>**javax.swing.plaf.basic**<br>**javax.swing.plaf.metal**<br>**javax.swing.plaf.multi** | *The Java Swing Pluggable-Look-and-Feel Packages.*<br>Estos paquetes contienen clases y una interfaz que se utilizan para cambiar la apariencia visual de una GUI basada en Swing, entre la apariencia visual de Java, la apariencia visual de Windows de Microsoft y la de UNIX Motif. El paquete también soporta el desarrollo de una apariencia visual personalizada para un programa en Java. |
| **javax.swing.table** | *The Java Swing Table Package.*<br>Este paquete contiene clases e interfaces para crear y manipular tablas al estilo de hojas de cálculo. |
| **javax.swing.text** | *The Java Swing Text Package.*<br>Este paquete contiene clases e interfaces para manipular texto basado en componentes GUI en Swing. |
| **javax.swing.text.html**<br>**javax.swing.text.html.parser** | *The Java Swing HTML Text Packages.*<br>Estos paquetes contienen una clase que proporciona soporte para construir editores de texto HTML. |

**Figura 25.10** Paquetes de la API de Java. (Parte 3 de 4.)

| Paquete | Descripción |
|---|---|
| **javax.swing.text.rtf** | *The Java Swing RTF Text Package*.<br>Este paquete contiene una clase que proporciona soporte para construir editores que soportan un rico formato de texto. |
| **javax.swing.tree** | *The Java Swing Tree Package*.<br>Este paquete contiene clases e interfaces para crear y manipular árboles de expansión de componentes GUI. |
| **javax.swing.undo** | *The Java Swing Undo Package*.<br>Este paquete contiene clases e interfaces que soportan el proporcionar capacidades de hacer y deshacer a un programa en Java. |
| **org.omg.CORBA**<br>**org.omg.CORBA.DynAnyPackage**<br>**org.omg.CORBA.ORBPackage**<br>**org.omg.CORBA.portable**<br>**org.omg.CORBA.TypeCodePackage**<br>**org.omg.CosNaming**<br>**org.omg.CosNaming.NamingContextPackage** | *The Object Management Group (OMG) CORBA Packages*.<br>Estos paquetes contienen clases e interfaces que implementan APIs CORBA de OMG que permiten a un programa en Java comunicarse con programas escritos en otros lenguajes de programación, de manera similar a cuando se utilizan los paquetes RMI de Java para comunicación entre programas en Java. |

**Figura 25.10** Paquetes de la API de Java. (Parte 4 de 4.)

## 25.6 Generación de números aleatorios

Ahora veremos nuevamente la simulación y los juegos (vea el capítulo 5). Como C, Java proporciona al programador los métodos para generar números aleatorios. En esta sección y en la siguiente, desarrollaremos versiones en Java de los programas de juegos que escribimos anteriormente en C.

Como recordará, en C utilizamos la función **rand** para generar números aleatorios. La función **rand** devolvía un valor entero entre **0** y la constante simbólica **RAND_MAX**. Los números aleatorios se generan de manera diferente en Java. La clase **Math** proporciona el método ***random***. Considere la siguiente instrucción:

```
double valorAleatorio = Math.random()
```

El método **random** genera un valor **double** mayor o igual que 0.0, pero menor que 1.0. Si **random** realmente produce valores al azar, todo valor mayor o igual que 0.0, pero menor que 1.0, tiene una *probabilidad* igual de ser elegido cada vez que se llama a **random**.

El rango de valores producidos directamente por **random**, con frecuencia es diferente de lo que se necesita en una aplicación específica. Por ejemplo, un programa que simula el tiro de un dado de seis lados requeriría números aleatorios en el rango de 1 a 6. Un programa que al azar predice el siguiente tipo de nave espacial (fuera de cuatro posibilidades) que volará a través del horizonte en un juego de video requeriría enteros aleatorios en el rango de 1 a 4.

Para mostrar **random**, desarrollemos una versión en Java del programa del capítulo 5 que simula 20 tiros de un dado y que imprime el valor de cada tiro. Utilicemos el operador de multiplicación (*****) junto con **random** de la siguiente manera

```
(int) ( Math.random() *6 )
```

para producir enteros en el rango de 0 a 5. Recordará que en C utilizamos el operador módulo (**%**) para *escalar* el valor de retorno de **rand**. Debido a que el método **random** de Java devuelve un valor **double** mayor o igual que 0.0, pero menor que 1.0, debemos multiplicar el número aleatorio por un factor de escala (en este

caso, 6) para escalar correctamente. El operador entero de conversión de tipo se utiliza para truncar la parte flotante (la parte que se encuentra después del número decimal) de cada valor producido por la expresión anterior. Después *desplazamos* el rango de números producidos sumando 1 al resultado anterior, como en

```
1 + (int) ( Math.random() * 6 )
```

La figura 25.11 confirma que los resultados se encuentran en el rango de 1 a 6.

```
// Figura 25.11: EntAleatorio.java
// Enteros aleatorios escalados y desplazados
import javax.swing.JOptionPane;

public class EntAleatorio {
   public static void main( String args[] )
   {
      int valor;
      String salida = "";

      for ( int i = 1; i <= 20; i++ ) {
         valor = 1 + (int) ( Math.random() * 6 );
         salida += valor + "  ";

         if ( i % 5 == 0 )
            salida += "\n";
      }

      JOptionPane.showMessageDialog( null, salida,
         "20 números aleatorios del 1 al 6",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin de main
} // fin de la clase EntAleatorio
```

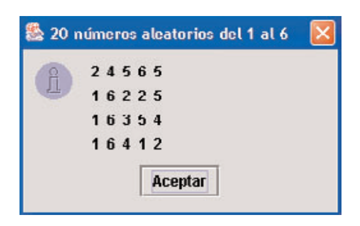



**Figura 25.11** Enteros aleatorios escalados y desplazados.

Para mostrar que estos números aparecen con aproximadamente la misma posibilidad, simulemos 6000 tiros de un dado con el programa de la figura 25.12. Cada entero de 1 a 6 debe aparecer aproximadamente 1000 veces.

```
// Figura 25.12: TiraDados.java
// Tira 6000 veces un dado de seis lados
import javax.swing.*;

public class TiraDados {
   public static void main( String args[] )
   {
      int frecuencia1 = 0, frecuencia2 = 0,
```

**Figura 25.12** Tiro de un dado 6000 veces. (Parte 1 de 2.)

```
           frecuencia3 = 0, frecuencia4 = 0,
           frecuencia5 = 0, frecuencia6 = 0, cara;

      // resume los resultados
      for ( int tiro = 1; tiro <= 6000; tiro++ ) {
         cara = 1 + (int) ( Math.random() * 6 );

         switch ( cara ) {
            case 1:
               ++frecuencia1;
               break;
            case 2:
               ++frecuencia2;
               break;
            case 3:
               ++frecuencia3;
               break;
            case 4:
               ++frecuencia4;
               break;
            case 5:
               ++frecuencia5;
               break;
            case 6:
               ++frecuencia6;
               break;
         } // fin de switch
      } // fin de for

      JTextArea areaSalida = new JTextArea( 7, 10 );

      areaSalida.setText(
         “Cara\tFrecuencia” +
         “\n1\t” + frecuencia1 +
         “\n2\t” + frecuencia2 +
         “\n3\t” + frecuencia3 +
         “\n4\t” + frecuencia4 +
         “\n5\t” + frecuencia5 +
         “\n6\t” + frecuencia6 );

      JOptionPane.showMessageDialog( null, areaSalida,
         “Lanzando 6000 veces un dado”,
         JOptionPane.INFORMATION_MESSAGE );
      System.exit( 0 );
   } // fin de main
} // fin de la clase TiraDados
```

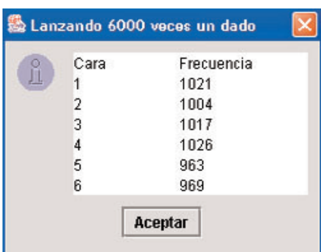



**Figura 25.12** Tiro de un dado 6000 veces. (Parte 2 de 2.)

Como muestra la salida del programa, al escalar y al desplazar hemos utilizado el método **`random`** para simular de manera real el tiro de un dado. El ciclo **`for`** de la línea 13 itera 6000 veces. Durante cada iteración del ciclo, la línea 14 produce un valor entre 1 y 6. La estructura **`switch`** anidada de la línea 16 utiliza el valor **`cara`** que se eligió al azar como su expresión de control. Basándose en el valor de **`cara`**, una de las seis variables contadores se incrementa durante cada iteración del ciclo. Observe que *no* se proporciona ningún caso **`default`** en la estructura **`switch`**. Después de que estudiemos los arreglos en las secciones 25.9 y 25.10, mostraremos cómo remplazar toda la estructura **`switch`** de este programa con una instrucción de una sola línea. Ejecute el programa varias veces y observe los resultados. Vea que se obtiene una secuencia *diferente* de números aleatorios cada vez que se ejecuta el programa, por lo que los resultados de éste deben variar.

Anteriormente mostramos cómo escribir una sola instrucción para simular el tiro de un dado, por medio de la instrucción

```
cara = 1 + (int) ( Math.random() * 6 );
```

la cual siempre asigna un entero (al azar) a la variable **`cara`**, en el rango 1≥ **`cara`** ≤6. Observe que el ancho de este rango (es decir, el número de enteros consecutivos en el rango) es 6, y que el número inicial del rango es 1. Considerando la instrucción anterior, vemos que el ancho del rango es determinado por el número utilizado para escalar **`random`** con el operador de multiplicación (es decir, 6), y que el número inicial del rango es igual que el número (es decir, 1) sumado a **`(int) ( Math.random() *`** 6 **`)`**. Podemos generalizar este resultado de la siguiente forma:

```
n = a + (int) ( Math.random() * b );
```

donde **`a`** es el *valor de desplazamiento* (el cual es igual al primer número del rango deseado de enteros consecutivos) y **`b`** es el *factor de escalamiento* (el cual es igual al ancho del rango deseado de enteros consecutivos).

## 25.7 Ejemplo: Un juego de azar

Recuerde nuestro ejemplo del juego de "craps" del capítulo 5. Ahora presentamos una nueva versión del simulador de craps como un applet de Java, en la figura 25.13.

```
// Figura 25.13: Craps.java
// Craps
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Craps extends JApplet implements ActionListener {
 // variables constantes para el estado del juego
 final int GANA = 0, PIERDE = 1, CONTINUAR = 2;

// otras variables utilizadas en el programa
boolean primerTiro = true;      // verdadero si es el primer tiro
int sumaDeDados = 0;            // suma de los dados
int miPunto = 0;   // punto si no se gana/pierde en el primer tiro
int estadoDelJuego = CONTINUAR; // el juego aún no ha terminado

// componentes de la interfaz gráfica de usuario
JLabel etiquetaDado1, etiquetaDado2, etiquetaSuma, etiquetaPunto;
JTextField primerDado, segundoDado, suma, punto;
JButton tiro;

// inicializa los componentes de la interfaz gráfica de usuario
public void init()
```

**Figura 25.13** Programa para simular el juego de craps. (Parte 1 de 4.)

```
{
   Container c = getContentPane();
   c.setLayout( new FlowLayout() );

   etiquetaDado1 = new JLabel( "Dado 1" );
   c.add( etiquetaDado1 );
   primerDado = new JTextField( 10 );
   primerDado.setEditable( false );
   c.add( primerDado );

   etiquetaDado2 = new JLabel( "Dado 2" );
   c.add( etiquetaDado2 );
   segundoDado = new JTextField( 10 );
   segundoDado.setEditable( false );
   c.add( segundoDado );

   etiquetaSuma = new JLabel( "La suma es" );
   c.add( etiquetaSuma );
   suma = new JTextField( 10 );
   suma.setEditable( false );
   c.add( suma );

   etiquetaPunto = new JLabel( "El puntaje es" );
   c.add( etiquetaPunto );
   punto = new JTextField( 10 );
   punto.setEditable( false );
   c.add( punto );

   tiro = new JButton( "Tirar dados" );
   tiro.addActionListener( this );
   c.add( tiro );
} // fin del método init

// llama al método jugar, cuando se oprime el botón
public void actionPerformed( ActionEvent e )
{
   jugar();
} // fin del método actionPerformed

// procesa un tiro de los dados
public void jugar()
{
   if ( primerTiro ) {              // primer tiro de los dados
      sumaDeDados = tiroDados();

      switch ( sumaDeDados ) {
         case 7: case 11:         // gana en el primer tiro
            estadoDelJuego = GANA;
            punto.setText( "" );  // limpia el campo de texto de puntaje
            break;
         case 2: case 3: case 12: // pierde en el primer tiro
            estadoDelJuego = PIERDE;
            punto.setText( "" );  // limpia el campo de texto de puntaje
            break;
         default:                 // recuerda el puntaje
```

**Figura 25.13** Programa para simular el juego de craps. (Parte 2 de 4.)

```
            estadoDelJuego = CONTINUAR;
            miPunto = sumaDeDados;
            punto.setText( Integer.toString( miPunto ) );
            primerTiro = false;
            break;
      } // fin de switch
   } // fin de if
   else {
      sumaDeDados = tiroDados();

      if ( sumaDeDados == miPunto )     // gana por puntos
         estadoDelJuego = GANA;
      else
         if ( sumaDeDados == 7 )        // pierde por tirar 7
            estadoDelJuego = PIERDE;
   } // fin de else

 if ( estadoDelJuego == CONTINUAR )
    showStatus( "Tire de nuevo." );
 else {
    if ( estadoDelJuego == GANA )
       showStatus( "El jugador gana. " +
           "Haga clic en Tirar dados para jugar de nuevo." );
    else
        showStatus( "El jugador pierde. " +
           "Haga clic en Tirar dados para jugar de nuevo." );

     primerTiro = true;
  } // fin de else
 // fin del método jugar

   // tirar dados
   public int tiroDados()
   {
      int dado1, dado2, trabajaSuma;

      dado1 = 1 + ( int ) ( Math.random() * 6 );
      dado2 = 1 + ( int ) ( Math.random() * 6 );
      trabajaSuma = dado1 + dado2;

      primerDado.setText( Integer.toString( dado1 ) );
      segundoDado.setText( Integer.toString( dado2 ) );
      suma.setText( Integer.toString( trabajaSuma ) );

      return trabajaSuma;
   } // fin del método tiroDados
} // fin de la clase Craps
```

Un objeto **JLabel** Un objeto **JTextField** Un objeto **JButton**

**Figura 25.13** Programa para simular el juego de craps. (Parte 3 de 4.)

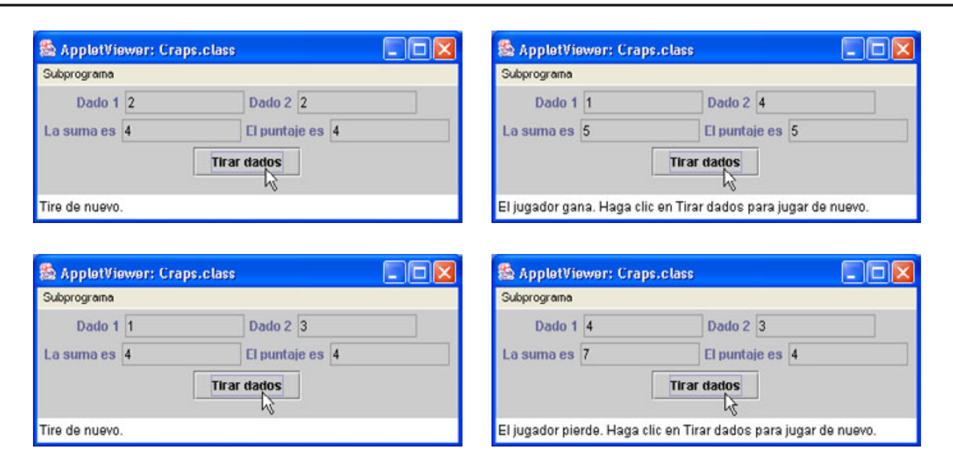



**Figura 25.13** Programa para simular el juego de craps. (Parte 4 de 4.)

Observe que así como en la versión de C del simulador, el jugador debe tirar dos dados en el primer tiro y en los subsiguientes. Cuando ejecute el applet, haga clic en el botón **Tirar dados**, para jugar. La esquina inferior izquierda de la ventana del **`appletviewer`** despliega los resultados de cada tiro. Las capturas de pantalla muestran cuatro ejecuciones separadas del applet (una en donde se gana y otra en donde se pierde en el primer tiro, y una en donde se gana y otra donde se pierde después del primer tiro).

Hasta el momento, todas las interacciones del usuario con aplicaciones y applets han sido a través de un diálogo de entrada (en el que el usuario podía escribir un valor de entrada para el programa), o a través de un diálogo de mensaje (en el que se desplegaba un mensaje para el usuario, y éste podía hacer clic en **Aceptar** para desechar el diálogo). Aunque éstas son formas válidas para recibir la entrada de un usuario y para desplegar resultados en un programa en Java, sus capacidades son bastante limitadas; un diálogo de entrada puede obtener sólo un valor a la vez por parte del usuario, y un diálogo de mensaje puede desplegar sólo un mensaje. Es mucho más común recibir múltiples entradas a la vez por parte del usuario (como la información sobre el nombre y la dirección del usuario), o desplegar muchas piezas de datos a la vez (en este ejemplo, los valores de los dados, la suma y el puntaje). Para comenzar nuestra introducción sobre interfaces de usuario más elaboradas, este programa ilustra dos nuevos conceptos sobre la interfaz gráfica de usuario; cómo adjuntar diversos componentes GUI a un applet, y la *manipulación de eventos* de la interfaz gráfica de usuario.

Las líneas 3 a 5

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

especifican al compilador en dónde localizar a las clases utilizadas en este applet. La primera **`import`** especifica que el programa utiliza clases del paquete **`java.awt`** (específicamente las clases **`Container`** y **`FlowLayout`**). La segunda **`import`** especifica que el programa utiliza clases del paquete **`java.awt.event`**. Este paquete contiene muchos tipos de datos que permiten a un programa procesar las interacciones de un usuario con la GUI de un programa. En este programa, utilizamos los tipos de datos **`ActionListener`** y **`ActionEvent`** del paquete **`java.awt.event`**. La última instrucción **`import`** especifica que el programa utiliza clases del paquete **`javax.swing`** (específicamente las clases **`JApplet`**, **`JLabel`**, **`JTextField`** y **`JButton`**).

Como dijimos antes, todo programa en Java se basa en al menos una definición de clase que amplía y mejora la definición de una clase existente, a través de la herencia. Recuerde que las applets se heredan de la clase **`JApplet`**. La línea 7

```
public class Craps extends JApplet implements ActionListener {
```

indica que la clase **`Craps`** se hereda de **`JApplet`** e implementa *(implements)* ***`ActionListener`***. Una clase puede heredar atributos y comportamientos (datos y métodos) de otra clase especificada a la derecha de la palabra reservada **`extends`**, en la definición de la clase. Además, una clase puede implementar una o más *in-*

*terfaces*. Una interfaz especifica uno o más comportamientos (es decir, métodos) *que usted debe definir* en la definición de la clase. La interfaz **ActionListener** especifica que esta clase *debe definir* un método con la primera línea

```
public void actionPerformed( ActionEvent e )
```

En este ejemplo, esta tarea del método es para procesar una interacción del usuario con el **JButton** (llamado **Tirar dados** en la interfaz de usuario). Cuando el usuario oprime el botón, éste método es invocado automáticamente, en respuesta a la interacción del usuario. A este proceso se le conoce como *manipulación de eventos*. El *evento* es la interacción del usuario (quien oprime el botón). El *manipulador de eventos* es el método **actionPerformed**, al cual se invoca automáticamente en respuesta al evento. Un poco más adelante explicaremos los detalles de esta interacción y del método **actionPerformed**. El capítulo 27 explica con detalle las interfaces. Por ahora, imite las características que ilustramos que soporten la manipulación de eventos de los componentes GUI que presentamos.

Este juego está razonablemente involucrado. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro. La línea 9 del programa

```
final int GANA = 0, PIERDE = 1, CONTINUAR = 2;
```

crea variables que definen los tres estados de un juego de craps; juego ganado, juego perdido o continuar el tiro de dados. La palabra reservada ***final***, al principio de la declaración, indica que éstas son *variables constantes*. Las variables constantes deben inicializarse una vez, antes de que se utilicen, y no pueden modificarse después. Con frecuencia, a las variables constantes se les conoce como *constantes nombradas* o *variables de sólo lectura*.



**Error común de programación 25.12**

*Después de que se inicializó una variable **final**, intentar asignar otro valor a esa variable es un error de sintaxis.*



**Buena práctica de programación 25.5**

*Sólo utilice letras mayúsculas (con guiones bajos entre las palabras) en los nombres de variables **final**. Esto hace que las constantes resalten en un programa.*



**Buena práctica de programación 25.6**

*Utilizar variables **final** con nombres descriptivos, en lugar de utilizar constantes enteras (como 2), hace que los programas sean legibles.*

Las líneas 12 a 15

```
boolean primerTiro = true;  // verdadero si es el primer tiro
int sumaDeDados = 0; // suma de los dados
int miPunto = 0;     // punto si no se gana/pierde en el primer tiro
int estadoDelJuego = CONTINUAR;   // el juego aún no ha terminado
```

declaran diversas variables de instancia que se utilizan a lo largo del applet **Craps**. La variable **primerTiro** indica si el siguiente tiro de los dados es el primero del juego actual. La variable **sumaDeDados** mantiene la suma de los dados correspondiente al último tiro. La variable **miPunto** almacena el "punto", si el jugador no gana o pierde en el primer tiro. La variable **estadoDelJuego** da seguimiento al estado actual del juego (**GANA**, **PIERDE** o **CONTINUAR**).

Las líneas 18 a 20

```
JLabel etiquetaDado1, etiquetaDado2, etiquetaSuma, etiquetaPunto;
JTextField primerDado, segundoDado, suma, punto;
JButton tiro;
```

declara referencias hacia los componentes GUI utilizados en la interfaz gráfica de usuario de este applet. Las referencias **etiquetaDado1**, **etiquetaDado2**, **etiquetaSuma** y **etiquetaPunto** se refieren a objetos ***JLabel***. Una **JLabel** contiene una cadena de caracteres a desplegar en la pantalla. Por lo general, una **JLabel** indica el propósito de otro elemento de la interfaz gráfica de usuario en la pantalla. En las capturas de pantalla de la figura 25.13, los objetos **JLabel** son el texto que se encuentra a la izquierda de cada rectángulo en las primeras dos filas de la interfaz de usuario. Las referencias **primerDado**, **segundoDado**, **su-**

**ma** y **punto** se refieren a objetos ***JTextField***. Los **JTextField** se utilizan para obtener una sola línea de información desde el teclado por parte del usuario, o para desplegar información en la pantalla. En las capturas de pantalla de la figura 25.13, los objetos **JTextField** son los rectángulos que se encuentran a la derecha de cada **JLabel**, en las dos primeras filas de la interfaz de usuario. La referencia **tiro** se refiere a un objeto ***JButton***. Cuando el usuario oprime un **JButton**, por lo general el programa responde realizando una tarea (en este ejemplo, tirando los dados). El objeto **JButton** es el rectángulo que contiene las palabras **Tira dados**, en la parte inferior de la interfaz de usuario de la figura 25.13. En ejemplos anteriores ya utilizamos **JtextFields** y **JButtons**. Todo mensaje de diálogo y todo diálogo de entrada contenía un botón **Aceptar** para desechar el diálogo de mensaje, o para enviar la entrada del usuario al programa. Todo diálogo de entrada también contenía un **JTextField**, en el que el usuario escribía un valor de entrada.

El método **init** (línea 23) crea los objetos componentes GUI y los adjunta a la interfaz de usuario. La línea 25

```
Container c = getContentPane();
```

declara una referencia **c** de **Container**, y le asigna el resultado de una llamada al método **getContentPane**. Recuerde, el método **getContentPane** devuelve una referencia al panel de contenido del applet que puede utilizarse para adjuntar componentes GUI a la interfaz de usuario del applet.

La línea 26

```
c.setLayout( new FlowLayout() );
```

utiliza el método ***setLayout*** de **Container** para definir un *administrador de diseño* para la interfaz de usuario del applet. Los administradores de diseño se proporcionan para acomodar los componentes GUI en un Container, para efectos de presentación. Estos administradores determinan la posición y el tamaño de cada componente GUI adjunto al contenedor. Esto permite al programador concentrarse en la “apariencia visual” básica, y deja a los administradores de diseño el procesamiento de la mayoría de los detalles del diseño.

**FlowLayout** es el administrador de diseño más básico. Los componentes GUI se colocan en un **Container** de izquierda a derecha, en el orden en el que se adjuntan al **Container** por medio del método **add**. Cuando se alcanza el borde del contenedor, los componentes continúan en la siguiente línea. La instrucción anterior crea un nuevo objeto de la clase **FlowLayout**, y lo pasa al método **setLayout**. En general, el diseño se establece antes de que cualquier componente GUI se agregue al **Container**.

[*Nota:* Cada **Container** puede tener sólo un administrador de diseño a la vez (**Containers** separados en el mismo programa pueden tener diferentes administradores de diseño). La mayoría de los ambientes de programación en Java proporcionan herramientas de diseño GUI que ayudan al programador a diseñar de manera gráfica una GUI, y después automáticamente se escribe código Java para crear la GUI. Algunos de estos diseñadores GUI también permiten al programador utilizar los administradores de diseño. El capítulo 29 explica diversos administradores de diseño, que permiten un control más preciso sobre el diseño de los componentes GUI.]

Las líneas 28 a 32, 34 a 38, 40 a 44, y 46 a 50 crean un par de **JLabel** y **JTextField**, y lo adjuntan a la interfaz de usuario. Estas líneas son muy parecidas, por lo que no concentraremos en las líneas 28 a 32.

```
etiquetaDado1 = new JLabel( “Dado 1” );
c.add( etiquetaDado1 );
primerDado = new JTextField( 10 );
primerDado.setEditable( false );
c.add( primerDado );
```

La línea 28 crea un nuevo objeto **JLabel**, lo inicializa con la cadena **“Dado 1”**, y asigna el objeto a la referencia **etiquetaDado1**. Esto etiqueta al **JTextField primerDado** correspondiente en la interfaz de usuario, por lo que el usuario puede determinar el propósito del valor desplegado en **primerDado**. La línea 29 adjunta la **JLabel** a la que **etiquetaDado1** hace referencia en el panel de contenido del applet. La línea 30 crea un nuevo objeto **JTextField**, lo inicializa para que sea de 10 caracteres de ancho, y asigna el objeto a la referencia **primerDado**. Este **JTextField** desplegará el valor del primer dado después de cada tiro de dados. La línea 31 utiliza el método **setEditable** de **JTextField** con el argumento **false** para indicar que el usuario no debe poder escribir en el **JTextField** (es decir, hace que el **JTextField** sea *ineditable*). Un **JTextField** no editable tiene de manera predeterminada un fondo gris (como se aprecia en

los diálogos de entrada). La línea 32 adjunta el **JTextField** a donde hace referencia **primerDado** en el panel de contenido del applet.

La línea 52

```
tiro = new JButton( "Tirar dados" );
```

crea un nuevo objeto **JButton**, lo inicializa con la cadena **"Tira dados"** (esta cadena aparecerá en la parte inferior), y asigna el objeto a la referencia **tiro**.

La línea 53

```
tiro.addActionListener( this );
```

especifica que *este* (***this***) applet debe *escuchar* los eventos de **tiro** de **JButton**. La palabra reservada **this** permite al applet hacer referencia a sí mismo (en el capítulo 26 explicaremos con detalle a **this**). Cuando el usuario interactúa con un componente GUI, se envía un *evento* al applet. Los eventos GUI son mensajes que indican que el usuario del programa interactuó con uno de los componentes GUI del programa. Por ejemplo, cuando en este programa oprime el **JButton tiro**, se envía un evento al applet que indica que el usuario oprimió el botón. Esto le indica al applet que el *usuario realizó una acción* en el **JButton**, y automáticamente llama al método **actionPerformed** para que procese la interacción del usuario.

A este estilo de programación se le conoce como *programación manejada por eventos*; el usuario interactúa con un componente GUI, al programa se le notifica el evento y lo procesa. La interacción del usuario con la GUI "maneja" el programa. Los métodos que son llamados cuando ocurre un evento también son conocidos como *métodos para manejo de eventos*. Cuando ocurre un evento GUI en un programa, Java crea un objeto que contenga la información sobre el evento que ocurrió, y *automáticamente llama* a un método para manejo de eventos apropiado. Antes de que pueda procesarse cualquier evento, cada componente GUI debe saber cuál objeto del programa define el método para manejo de eventos que se llamará cuando ocurra un evento. En la línea 53, se utiliza el método ***addActionListener*** de **JButton** para decirle a **tiro** que el applet (**this**) puede escuchar *eventos de acción*, y define un método **actionPerformed**. A esto se le conoce como *registro del manipulador de eventos* con el componente GUI (también quisiéramos llamarlo la línea que *empieza a escuchar*, ya que el applet ahora está escuchando los eventos del botón). Para responder a un evento de acción, debemos definir una clase que implemente un **ActionListener** (esto requiere que la clase también defina un método **actionPerformed**) y debemos registrar el manipulador de eventos con el componente GUI. Por último, la última línea de **init** adjunta el **JButton** al que **tiro** hace referencia en el panel de contenido del applet, con lo que se completa la interfaz de usuario.

El método **actionPerformed** (línea 58) es uno de diversos métodos que procesan las interacciones entre el usuario y los componentes GUI. La primera línea del método

```
public void actionPerformed( ActionEvent e )
```

indica que **actionPerformed** es un método **public** que devuelve nada (**void**) cuando completa su tarea. Cuando se llama automáticamente, el método **actionPerformed** recibe un argumento (un **ActionEvent**), en respuesta a una acción realizada por el usuario sobre un componente GUI (en este caso, oprimir el **JButton**). El argumento **ActionEvent** contiene información acerca de la acción que ocurrió.

Definimos un método **tiraDados** (línea 111) para tirar los dados y para calcular y desplegar su suma. El método **tiraDados** se define una vez, pero se le llama desde dos lugares del programa (líneas 67 y 87). El método **tiraDados** no toma argumentos, por lo que tiene una lista de parámetros vacía. El método **tiraDados** devuelve la suma de los dos dados, por lo que en el encabezado del método se indica un tipo de retorno **int**.

El usuario hace clic en el botón **"Tira Dados"** para realizar su tiro. Esto invoca al método **actionPerformed** (línea 58) del applet, el cual después invoca al método **jugar** (definido en la línea 64). El método **jugar** verifica la variable **booleana primerTiro** (línea 66) para determinar si es **true** o **false**. Si es **true**, éste es el primer tiro del juego. La línea 67 llama a **tiraDados** (definido en la línea 111), el cual escoge dos valores al azar entre 1 y 6, despliega el valor del primer dado, el del segundo dado y la suma de los dos en los tres primeros **JTextFields**, y devuelve la suma de los dados. Observe que los valores enteros se convierten en **Strings** con el método **static Integer.toString**, ya que los **JTextStrings** sólo

pueden desplegar cadenas. Después del primer tiro, la estructura **switch** anidada de la línea 69 determina si se ganó o se perdió el juego, o si el juego debe continuar con otro tiro. Después del primer tiro, si el juego no terminó, la suma se guarda en **miPunto** y se despliega en el **JTextField punto**.

El programa continúa con la estructura **if/else** anidada de la línea 96, la cual utiliza el método **showStatus** para desplegar en la barra de estado del **appletviewer**

**Tira de nuevo.**

si el **estadoDelJuego** es igual que **CONTINUAR** y

**El jugador gana. Haga clic en Tira Dados para jugar otra vez.**

si el **estadoDelJuego** es igual que **GANA** y

**El jugador pierde. Haga clic en Tira Dados para jugar otra vez.**

si el **estadoDelJuego** es igual que **PIERDE**. El método **showStatus** recibe un argumento **String** y lo despliega en la barra de estado del **appletviewer** o navegador. Si se ganó o se perdió el juego, la línea 106 establece en **true** a **primerTiro**, para indicar que el siguiente tiro de dados es el primero del siguiente juego.

El programa entonces espera que el usuario nuevamente haga clic en el botón **"Tira Dados"**. Cada vez que el usuario presione este botón, el método **actionPerformed** invoca al método **jugar** y el método **tiraDados** es llamado para producir una nueva **suma**. Si la **suma** coincide con **miPunto**, el **estadoDeJuego** se establece en **GANA**, la estructura **if/else** de la línea 96 se ejecuta y el juego se completa. Si la **suma** es igual que **7**, **estadoDelJuego** se establece en **PIERDE**, la estructura **if/else** de la línea 96 se ejecuta y el juego se completa. Al hacer clic en el botón **"Tira Dados"** se inicia un nuevo juego. A lo largo del programa, los cuatro **JTextFields** se actualizan con los nuevos valores de los dados y la suma en cada tiro, y el **JTextField** punto se actualiza cada vez que inicia un nuevo juego.

## 25.8 Métodos de la clase JApplet

Hasta este punto del texto hemos escrito muchos applets, pero aún no hemos explicado los métodos clave de la clase **JApplet** que son llamados automáticamente durante la ejecución de un applet. La figura 25.14 lista los métodos clave de la clase **JApplet**, cuándo se les llama, y el propósito de cada uno.

Estos métodos de **JApplet** son definidos por la API de Java para que no hagan cosa alguna, a menos que usted proporcione una definición en la definición de la clase de su applet. Si quisiera utilizar uno de estos métodos en un applet que está definiendo, debe definir la primera línea del método como muestra la figura 25.14. De lo contrario, el método no será llamado automáticamente durante la ejecución del applet.

| Método | Cuándo se llama al método y su propósito |
|---|---|
| **public void init()** | A este método lo llama una vez el **appletviewer** o el navegador cuando se carga un applet para su ejecución. Éste realiza la inicialización de un applet. Las acciones típicas que se realizan aquí son la inicialización de variables de instancia y de componentes GUI del applet, la carga de sonidos a reproducir o imágenes a desplegar (capítulo 30), y la creación de subprocesos. |
| **public void start()** | Este método es llamado después de que el método **init** completa su ejecución, y cada vez que el usuario del navegador regresa a la página HTML en la que reside el applet (después de explorar otra página HTML). Este método realiza cualquier tarea que deba completarse cuando el applet se carga por primera vez en el navegador, y que deba realizarse cada vez que la página HTML en la que reside el applet se vuelva a visitar. Las acciones típicas que se realizan aquí incluyen iniciar una animación (capítulo 30), e iniciar otros subprocesos de ejecución. |

**Figura 25.14** Métodos de **JApplet** que se llaman automáticamente durante la ejecución de un applet. (Parte 1 de 2.)

| Método | Cuándo se llama al método y su propósito |
|---|---|
| **`public void paint( Graphics g )`** | |
| | Este método es llamado para dibujar en el applet, después de que el método **`init`** completa su ejecución y después de que el método **`start`** ha iniciado su ejecución. También se le llama automáticamente cada vez que el applet necesita repintarse. Por ejemplo, si el usuario del navegador cubre el applet con otra ventana abierta en la pantalla, entonces descubre el applet, y se llama al método **`paint`**. Las acciones típicas realizadas aquí involucran el dibujo con el objeto **`g`** de **`Graphics`**, el cual es pasado al método **`paint`**. |
| **`public void stop()`** | |
| | Este método es llamado cuando el applet debe detener su ejecución; normalmente cuando el usuario del navegador abandona la página HTML en la que el applet reside. Este método realiza cualquier tarea necesaria para suspender la ejecución del applet. Las acciones típicas realizadas aquí son detener la ejecución de animaciones y subprocesos. |
| **`public void destroy()`** | |
| | Este método es llamado cuando el applet está siendo removido de la memoria; normalmente cuando el usuario del navegador abandona la sesión de navegación. Este método realiza cualquier tarea necesaria para destruir recursos asignados al applet. |

**Figura 25.14** Métodos de **`JApplet`** que se llaman automáticamente durante la ejecución de un applet. (Parte 2 de 2.)



**Error común de programación 25.13**

*Proporcionar una definición para uno de los métodos de **`JApplet`** **`init`**, **`start`**, **`paint`**, **`stop`** o **`destroy`**, que no coincida con los encabezados de los métodos que muestra la figura 25.14, dará como resultado un método que no será llamado automáticamente durante la ejecución del applet.*

El método ***`repaint`*** también es de interés para muchos programadores de applets. El método **`paint`** del applet por lo general se invoca automáticamente. ¿Qué sucedería si quisiera cambiar la apariencia del applet, en respuesta a interacciones del usuario con el applet? En tales situaciones, podría desear llamar directamente a **`paint`**. Sin embargo, para llamar a **`paint`**, debemos pasarle el parámetro **`Graphics`** que espera. Esto implica un problema para nosotros. Nosotros no tenemos un objeto **`Graphics`** a nuestra disposición para pasarlo a **`paint`** (en el capítulo 30 explicaremos este asunto). Por esta razón, se le proporciona el método **`repaint`**. La instrucción

```
repaint();
```

invoca otro método llamado ***`update`***, y le pasa el objeto **`Graphics`** por usted. El método **`update`** borra cualquier dibujo que se hubiera hecho anteriormente en el applet, después invoca al método **`paint`** y le pasa el objeto **`Graphics`** por usted. En el capítulo 30 explicamos con detalle los métodos **`repaint`** y **`update`**.

## 25.9 Declaración y asignación de arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de los elementos y utiliza el operador **`new`** para asignar dinámicamente el número de elementos requeridos por cada arreglo. Los arreglos se asignan con **`new`**, debido a que éstos se consideran como objetos, y todos los objetos deben crearse con **`new`**. Para asignar 12 elementos al arreglo entero **`c`**, se utiliza la declaración

```
int c[] = new int[12];
```

La instrucción anterior también puede realizarse en dos pasos, de la siguiente manera:

```
int c[];             // declara el arreglo
c = new int[ 12 ];   // asigna el arreglo
```

Cuando los arreglos se asignan, los elementos se inicializan en cero, si se trata de variables de tipos de datos primitivos, en **`false`**, si se trata de variables **`boolean`**, o en **`null`**, si se trata de referencias (de cualquier tipo que no sea primitivo).



**Error común de programación 25.14**

*A diferencia de C o de C++, el número de elementos en el arreglo nunca se especifica entre corchetes después del nombre del arreglo en una declaración. La declaración* **`int c[12];`** *ocasiona un error de sintaxis.*

La memoria puede reservarse para diversos arreglos con una sola declaración. La siguiente declaración reserva 100 elementos para el arreglo **`String b`**, y 27 elementos para el arreglo **`String x`**:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

Cuando se declara un arreglo, al principio de la declaración podemos combinar el tipo del arreglo y los corchetes, para indicar que todos los identificadores en la declaración representan arreglos, como en

```
double[] arreglo1, arreglo2;
```

la cual declara tanto el **`arreglo1`** como el **`arreglo2`** como valores **`double`**. Como mostramos anteriormente, la declaración e inicialización de un arreglo pueden combinarse en la declaración. La siguiente declaración reserva 10 elementos para el **`arreglo1`** y 20 elementos para el **`arreglo2`**:

```
double[] arreglo1 = new double[ 10 ], arreglo2 = new double[ 20 ];
```

Los arreglos pueden declararse para que contengan cualquier tipo de dato. Es importante recordar que en un arreglo de tipo de datos primitivos, cada elemento contiene un valor del tipo de datos declarado para el arreglo. Sin embargo, en un arreglo de tipo no primitivo, cada elemento es una referencia hacia un objeto del tipo del arreglo. Por ejemplo, cada elemento de un arreglo **`String`** es una referencia hacia una **`String`** que tiene de manera predeterminada el valor **`null`**.

## 25.10 Ejemplos del uso de arreglos

La aplicación de la figura 25.15 utiliza el operador **`new`** para asignar dinámicamente un arreglo de 10 elementos que se inicializan en cero, y después imprime el arreglo en formato tabular.

```
// Figura 25.15: InicArreglo.java
// inicializa un arreglo
import javax.swing.*;

public class InicArreglo {
   public static void main( String args[] )
   {
      String salida = "";
      int n[];              // declara una referencia a un arreglo

      n = new int[ 10 ];  // asigna dinámicamente el arreglo

      salida += "Subindice\tValor\n";

      for ( int i = 0; i < n.length; i++ )
         salida += i + "\t" + n[ i ] + "\n";

      JTextArea areaSalida = new JTextArea( 11, 10 );
      areaSalida.setText( salida );

      JOptionPane.showMessageDialog( null, areaSalida,
```

**Figura 25.15** Inicialización en cero de los elementos de un arreglo. (Parte 1 de 2.)

```
22             "Inicializa un arreglo con valores int",
23             JOptionPane.INFORMATION_MESSAGE );
24
25          System.exit( 0 );
26       } // fin de main
27    } // fin de la clase InicArreglo
```

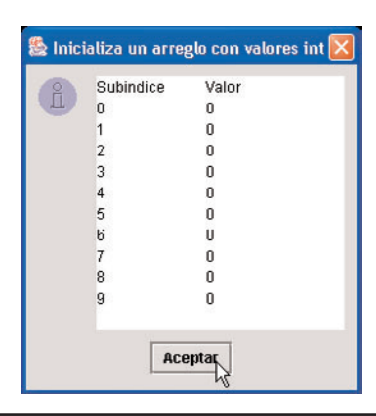



**Figura 25.15** Inicialización en cero de los elementos de un arreglo. (Parte 2 de 2.)

La línea 9 declara a **`n`** como una referencia capaz de referirse a un arreglo de enteros. La línea 11 asigna los 10 elementos del arreglo con **`new`**, e inicializa la referencia. La línea 13 agrega a **`String salida`** los encabezados para las columnas de la salida desplegada por el programa.

Las líneas 15 y 16

```
for ( int i = 0; i < n.lenght; i++ )
   salida += i + "\t" + n[ i ] + "\n";
```

utilizan una estructura **`for`** para construir la **`String`** de **`salida`** que se desplegará en un **`JTextArea`** de un diálogo de mensaje. Observe el uso de la cuenta basada en cero (recuerde, los subíndices inician en 0), por lo que el ciclo puede acceder a cada elemento del arreglo. Además, observe la expresión **`n.lenght`** en la condición de la estructura **`for`** para determinar la longitud del arreglo. En este ejemplo, la longitud del arreglo es 10, por lo que el ciclo continúa en ejecución mientras el valor de la variable de control **`i`** sea menor que 10. Para un arreglo de 10 elementos, los valores de los subíndices van de 0 a 9, por lo que utilizar el operador de menor que (**<**) garantiza que el ciclo no intentará acceder a un elemento que se encuentre más allá del final del arreglo. Los elementos de un arreglo pueden asignarse e inicializarse en la declaración del arreglo, colocando después de la declaración un signo de igual y una *lista de inicializadores* separada por comas entre llaves (**`{`** y **`}`**). En este caso, el tamaño del arreglo se determina por medio del número de elementos en la lista de inicialización. Por ejemplo, la instrucción

```
int n[] = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los subíndices **`0`**, **`1`**, **`2`**, **`3`** y **`4`**. Observe que la declaración anterior no requiere el operador **`new`** para crear el objeto arreglo; esto lo proporciona el compilador siempre que encuentra una declaración de arreglo que incluye una lista de inicialización.

La aplicación de la figura 25.16 inicializa un arreglo entero con 10 valores (línea 12) y lo despliega en formato tabular en un **`JTextArea`** de un diálogo de mensaje.

```
1  // Figura 25.16: InitArray.java
2  // inicializa un arreglo mediante una declaración
3  import javax.swing.*;
4
```

**Figura 25.16** Inicialización de los elementos de un arreglo por medio de una declaración. (Parte 1 de 2.)

```
public class InicArreglo {
   public static void main( String args[] )
   {
      String salida = "";

      // La lista de inicialización especifica el número de elementos y
      // el valor de cada elemento.
      int n[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      salida += "Subindice\tValor\n";

      for ( int i = 0; i < n.length; i++ )
         salida += i + "\t" + n[ i ] + "\n";

      JTextArea areaSalida = new JTextArea( 11, 10 );
      areaSalida.setText( salida );

      JOptionPane.showMessageDialog( null, areaSalida,
         "Inicializa un arreglo mediante una declaracion",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin de main
} // fin de la clase InicArreglo
```

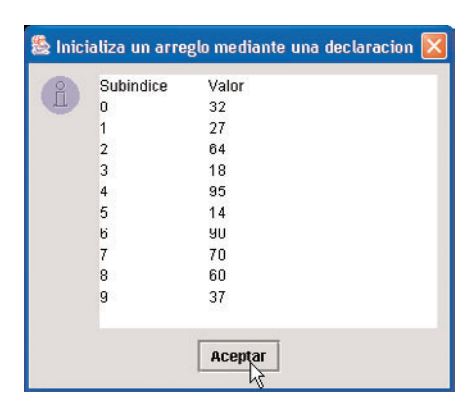



**Figura 25.16** Inicialización de los elementos de un arreglo por medio de una declaración. (Parte 2 de 2.)

La aplicación de la figura 25.17 inicializa los elementos del arreglo **s** de 10 elementos con los enteros pares **2**, **4**, **6**, ..., **20** y lo imprime en formato tabular. Estos números se generan multiplicando cada valor sucesivo del contador del ciclo por **2** y sumándole **2**.

```
// Figura 25.17: InicArreglo.java
// inicializa el arreglo n con los enteros pares de 2 a 20
import javax.swing.*;

public class InicArreglo {
   public static void main( String args[] )
   {
      final int TAMANIO_ARREGLO = 10;
      int n[];                         // referencia a un arreglo de enteros
```

**Figura 25.17** Generación de valores para colocarlos como elementos de un arreglo. (Parte 1 de 2.)

```
10         String salida = "";
11
12         n = new int[ TAMANIO_ARREGLO ];  // asigna el arreglo
13
14         // Establece los valores en el arreglo
15         for ( int i = 0; i < n.length; i++ )
16            n[ i ] = 2 + 2 * i;
17
18         salida += "Subindice\tValor\n";
19
20         for ( int i = 0; i < n.length; i++ )
21            salida += i + "\t" + n[ i ] + "\n";
22
23         JTextArea areaSalida = new JTextArea( 11, 10 );
24         areaSalida.setText( salida );
25
26         JOptionPane.showMessageDialog( null, areaSalida,
27            "Inicializa a numeros pares de 2 a 20",
28            JOptionPane.INFORMATION_MESSAGE );
29
30         System.exit( 0 );
31      } // fin de main
32   } // fin de la clase InicArreglo
```

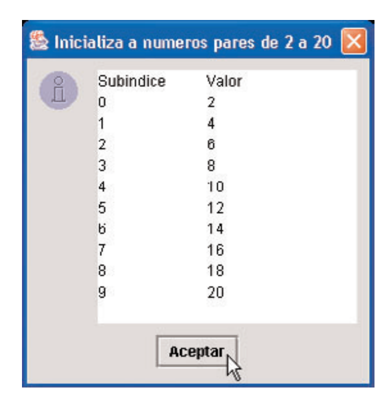



**Figura 25.17** Generación de valores para colocarlos como elementos de un arreglo. (Parte 2 de 2.)

La línea 8

```
final int TAMANIO_ARREGLO = 10;
```

utiliza el calificador **`final`** para declarar una variable constante llamada **`TAMANIO_ARREGLO`**, cuyo valor es **`10`**. Las variables constantes deben inicializarse antes de utilizarlas, y no pueden modificarse después. Si se intenta modificar una variable **`final`** después de declararla como muestra la instrucción anterior, el compilador despliega un mensaje como

```
Can't assign a value to a final variable
```

Si se intenta modificar una variable **`final`** después de que se declara, y después se inicializa en una instrucción separada, el compilador despliega un mensaje de error como

```
Can't assign a second value to a blank final variable
```

Si se intenta utilizar una variable local **`final`** antes de que se inicialice, el compilador despliega el mensaje de error

**`Variable`** *nombreVariable* **`may not have been initialized`**

```
1  // Figura 25.18: SumaArreglo.java
2  // Cálculo de la suma de los elementos de un arreglo
3  import javax.swing.*;
4
5  public class SumaArreglo {
6     public static void main( String args[] )
7     {
8        int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9        int total = 0;
10
11       for ( int i = 0; i < a.length; i++ )
12          total += a[ i ];
13
14       JOptionPane.showMessageDialog( null,
15          "Total de elementos en el arreglo: " + total,
16          "Suma de los elementos del arreglo",
17          JOptionPane.INFORMATION_MESSAGE );
18
19       System.exit( 0 );
20    } // fin de main
21 } // fin de la clase SumaArreglo
```

**Figura 25.18** Cálculo de la suma de los elementos de un arreglo.

Si se intenta utilizar una variable de instancia **`final`** antes de inicializarla, el compilador despliega el mensaje de error

> **`Blank final variable`** *'nombreVariable'* **`may not have been initialized. It must be assigned a value in an initializer, or in every constructor.`**

Las variables constantes también son conocidas como *constantes nombradas* o *variables de sólo lectura*. Con frecuencia se utilizan para hacer que un programa sea más legible. Observe que el término "constante variable" es un oxímoron (una contradicción en términos) como "chico grande" o "bien mal".



**Error común de programación 25.15**

*Asignar un valor a una constante variable después de inicializarla, es un error de sintaxis.*

La aplicación de la figura 25.18 suma los valores contenidos en el arreglo entero **`a`** de 10 elementos (declarado, asignado e inicializado en la línea 8). La instrucción (línea 12) en el cuerpo del ciclo **`for`** hace la totalización. Es importante recordar que los valores que se proporcionaron como inicializadores para el arreglo **`a`** normalmente se leerían en el programa. Por ejemplo, en un applet, el usuario podría introducir los valores a través de un **`JTextField`**, o en una aplicación los valores podrían leerse desde un archivo en disco.

Nuestro siguiente ejemplo utiliza arreglos para resumir los resultados de los datos obtenidos en una encuesta. Considere el problema:

> *Se les pidió a cuarenta estudiantes que calificaran la calidad de la comida de una cafetería para estudiantes en una escala de 1 a 10 (1 significa terrible, y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero y totalice los resultados de la encuesta.*

Ésta es una aplicación típica del procesamiento de arreglos (vea la figura 25.19). Querríamos resumir el número de respuestas de cada tipo (es decir, 1 a 10). El arreglo **`respuestas`** es un arreglo entero de 40 elementos que contiene las respuestas de los estudiantes a la encuesta. Utilizamos un arreglo **`frecuencia`** de 11 elementos

```
// Figura 25.19: EncEstudiantes.java
// Programa de encuesta a estudiantes
import javax.swing.*;

public class EncEstudiantes {
   public static void main( String args[] )
   {
      int respuestas[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
                           1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
                           6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
                           5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
      int frecuencia[] = new int[ 11 ];
      String salida = "";

      for ( int contestacion = 0;                    // inicializa
            contestacion < respuestas.length;        // condición
            contestacion++ )                         // incremento
         ++frecuencia[ respuestas[ contestacion ] ];

      salida += "Calificacion\tFrecuencia\n";

      for ( int calificacion = 1;
            calificacion < frecuencia.length;
            calificacion++ )
         salida += calificacion + "\t" + frecuencia[ calificacion ] + "\n";

      JTextArea areaSalida = new JTextArea( 11, 10 );
      areaSalida.setText( salida );

      JOptionPane.showMessageDialog( null, areaSalida,
         "Programa de encuesta a estudiantes",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin de main
} // fin de la clase EncEstudiantes
```

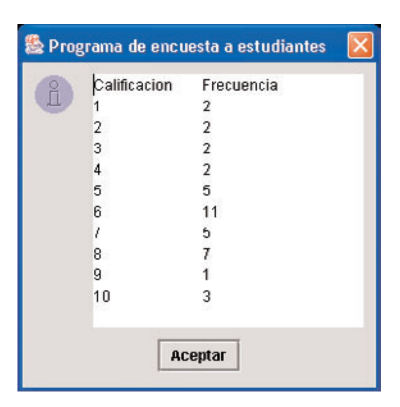



**Figura 25.19** Un programa de análisis de una encuesta sencilla a estudiantes.

para contar el número de ocurrencias de cada respuesta. Ignoramos el primer elemento, **`frecuencia[ 0 ]`**, ya que es más lógico hacer que la respuesta 1 incremente a **`frecuencia[ 1 ]`**, en lugar de a **`frecuencia[ 0 ]`**. Esto nos permite utilizar cada respuesta de manera directa como un subíndice del arreglo **`frecuencia`**. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta.

### Buena práctica de programación 25.7

*Esfuércese en favor de la claridad de sus programas. Algunas veces vale la pena sacrificar el uso más eficiente de la memoria o el tiempo de procesamiento, a favor de escribir programas más claros.*



### Tip de rendimiento 25.2

*Algunas veces las consideraciones de rendimiento se contraponen a las consideraciones de claridad.*

El ciclo `for` (líneas 15 a 18) toma las respuestas, una a la vez, del arreglo `respuestas` e incrementa uno de los 10 contadores del arreglo `frecuencia` (`frecuencia[ 1 ]` a `frecuencia[ 10 ]`). La instrucción clave del ciclo es

```
++frecuencia[ respuestas[ contestacion ] ];
```

Esta instrucción incrementa el contador `frecuencia` apropiado, de acuerdo con el valor de `respuestas[ contestacion ]`.

Consideremos diversas iteraciones del ciclo `for`. Cuando el contador `contestacion` es `0`, `respuestas[ contestacion ]` es el valor del primer elemento del arreglo `respuestas` (es decir, `1`), por lo que `++frecuencia[ respuestas[ contestacion ] ];` en realidad se interpreta como

```
++frecuencia[ 1 ];
```

la cual incrementa el elemento uno del arreglo. Al evaluar la expresión, comience con el valor que se encuentra en el conjunto de corchetes más internos (`contestacion`). Una vez que sepa el valor de `contestacion`, conecte el valor que se encuentra en la expresión y evalúe el siguiente conjunto externo de corchetes (`respuestas[ contestacion ]`). Después utilice ese valor como el subíndice del arreglo `frecuencia` para determinar cuál contador incrementar.

Cuando `contestacion` es `1`, `respuestas[ contestacion ]` es el valor del segundo elemento del arreglo `respuestas` (es decir, `2`), por lo que `++frecuencia[ respuestas [ contestacion ] ];` en realidad se interpreta como

```
++frecuencia[ 2 ];
```

la cual incrementa el elemento dos del arreglo (el tercer elemento del arreglo).

Cuando `contestacion` es `2`, `respuestas[ contestacion ]` es el valor del tercer elemento del arreglo `respuestas` (es decir, `6`), por lo que `fecuencia[ respuestas[ contestacion ]];` en realidad se interpreta como

```
++frecuencia[ 6 ];
```

la cual incrementa el elemento seis del arreglo (el séptimo elemento del arreglo), y así sucesivamente. Observe que independientemente del número de respuestas procesadas de la encuesta, sólo se requiere un arreglo de 11 elementos (sin tomar en cuenta el elemento cero) para resumir los resultados, ya que todos los valores de respuesta se encuentran entre 1 y 10, y los valores de los subíndices para un arreglo de 11 elementos van de 0 a 10. También observe que los resultados son correctos, debido a que los elementos del arreglo `frecuencia` se inicializaron automáticamente en cero cuando el arreglo se asignó con `new`.

Si los datos contuvieran valores inválidos, como 13, el programa intentaría sumar `1` a `frecuencia[ 13 ]`, lo que estaría fuera de los límites del arreglo. En C y en C++, el compilador permitiría tales referencias, y en tiempo de ejecución, el programa sobrepasaría el final del arreglo hacia donde creyera que se encuentra el elemento número 13, y sumaría 1 a lo que estuviera en esa ubicación de memoria. Esto podría potencialmente modificar otra variable del programa, o incluso podría resultar en una terminación prematura del programa. Java proporciona mecanismos para evitar el acceso a elementos que se encuentren fuera de los límites de un arreglo.



### Tip para prevenir errores 25.2

*Cuando se ejecuta un programa en Java, el intérprete de Java verifica los subíndices de los elementos del arreglo para asegurarse de que son válidos (es decir, todos los subíndices deben ser mayores o iguales que 0, y menores que la longitud del arreglo). Si hay un subíndice inválido, Java genera una excepción.*



### Tip para prevenir errores 25.3

*Las excepciones se utilizan para indicar que ocurrió un error en un programa. Éstas permiten que el programador se recupere del error y continúe con la ejecución del programa, en lugar de terminarlo de manera anormal. Cuando se hace una referencia inválida hacia un arreglo, se genera una excepción* **`ArrayIndexOutOfBoundsException`**.



### Error común de programación 25.16

*Hacer referencia a un elemento que se encuentra fuera de los límites de un arreglo, es un error lógico.*



### Tip para prevenir errores 25.4

*Cuando se hace un ciclo a través de un arreglo, los subíndices de éste nunca deben ir por debajo de 0 y siempre deben ser menores que el número total de elementos del arreglo (uno menos que el tamaño de éste). Asegúrese de que la condición de terminación del ciclo evite el acceso a elementos fuera de este rango.*



### Tip para prevenir errores 25.5

*Los programas deben validar que todos los valores de entrada sean correctos para prevenir que información errónea afecte a los cálculos de un programa.*

Nuestra siguiente aplicación (figura 25.20) lee los números de un arreglo y grafica la información en un gráfico de barras (o histograma); cada número se imprime, y después, a un lado de éstos, se despliega una barra consistente en los asteriscos que representen a ese número. El ciclo anidado **`for`** (líneas 13 a 18) agrega las barras a la **`String`** que se desplegará en el **`JTextArea areaSalida`** de un diálogo de mensaje. Observe la condición de terminación de ciclo de la estructura **`for`** interna de la línea 16 **`(j <= n[ i ])`**. Cada vez que se alcanza la estructura **`for`** interna, ésta cuenta de **`1`** hasta **`n[ i ]`**, por lo que utiliza un valor del arreglo **`n`** para determinar el valor final de la variable de control **`j`** y el número de asteriscos a desplegar.

```
1  // Figura 25.20: Histograma.java
2  // Programa de impresión de un histograma
3  import javax.swing.*;
4  
5  public class Histograma {
6     public static void main( String args[] )
7     {
8        int n[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9        String salida = "";
10 
11       salida += "Elemento\tValor\tHistograma";
12 
13       for ( int i = 0; i < n.length; i++ ) {
14          salida += "\n" + i + "\t" + n[ i ] + "\t";
15 
16          for ( int j = 1; j <= n[ i ]; j++ ) // imprime una barra
17             salida += "*";
18       } // fin de for
19 
20       JTextArea areaSalida = new JTextArea( 11, 30 );
21       areaSalida.setText( salida );
22 
23       JOptionPane.showMessageDialog( null, areaSalida,
24          "Programa de impresion de un histograma",
25          JOptionPane.INFORMATION_MESSAGE );
26 
27       System.exit( 0 );
28    } // fin de main
29 } // fin de la clase Histograma
```

**Figura 25.20** Un programa que imprime histogramas. (Parte 1 de 2.)

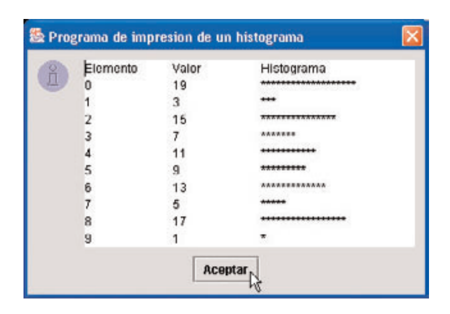



**Figura 25.20** Un programa que imprime histogramas. (Parte 2 de 2.)

La sección 25.6 indicó que existe un método más elegante para escribir el programa de tiro de dados de la figura 25.12. El programa tiraba 6000 veces un dado de seis lados. Una versión con arreglos de esta aplicación aparece en la figura 25.21. Las líneas 16 a 35 de la figura 25.12 se reemplazan con la línea 13 de este programa, la cual utiliza el valor aleatorio **`cara`** como el subíndice del arreglo **`frecuencia`**, para determinar cuál elemento debe incrementarse durante cada iteración del ciclo. El cálculo del número aleatorio de la línea 12 produce números entre 1 y 6 (los valores del dado de seis lados), por lo que el arreglo **`frecuencia`** debe ser lo suficientemente grande para permitir valores de subíndices de 1 a 6. El número más pequeño de elementos requerido para un arreglo que tenga estos valores de subíndices es de siete elementos (valores de subíndices de 0 a 6). En este programa, ignoramos el elemento 0 del arreglo **`frecuencia`**. Además, las líneas 18 y 19 de este programa reemplazan a las líneas 40 a 47 de la figura 25.12. Podemos realizar un ciclo a través del arreglo **`frecuencia`**, por lo que no tenemos que numerar cada línea de texto a desplegar en el **`JTextArea`**, como hicimos en la figura 25.12.

```
1  // Figura 25.21: TiraDado.java
2  // Tira los dados 6000 veces
3  import javax.swing.*;
4
5  public class TiraDado {
6     public static void main( String args[] )
7     {
8        int cara, frecuencia[] = new int[ 7 ];
9        String salida = "";
10
11       for ( int tiro = 1; tiro <= 6000; tiro++ ) {
12          cara = 1 + ( int ) ( Math.random() * 6 );
13          ++frecuencia[ cara ];
14       }
15
16       salida += "Cara\tFrecuencia";
17
18       for ( cara = 1; cara < frecuencia.length; cara++ )
19          salida += "\n" + cara + "\t" + frecuencia[ cara ];
20
21       JTextArea areaSalida = new JTextArea( 7, 10 );
22       areaSalida.setText( salida );
23
24       JOptionPane.showMessageDialog( null, areaSalida,
25          "Tira los dados 6000 veces",
26          JOptionPane.INFORMATION_MESSAGE );
```

**Figura 25.21** Programa para tirar dados por medio de arreglos, en lugar de la instrucción **`switch`**. (Parte 1 de 2.)

```

      System.exit( 0 );
   } // fin de main
} // fin de la clase TiraDado
```

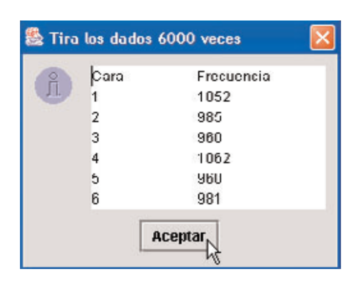



**Figura 25.21** Programa para tirar dados por medio de arreglos, en lugar de la instrucción **`switch`**. (Parte 2 de 2.)

## 25.11 Referencias y parámetros de referencias

Dos formas para pasar argumentos hacia los métodos (o funciones) en muchos lenguajes de programación (como C y C++) son las *llamadas por valor* y las *llamadas por referencia* (también conocidas como *pasar por valor* y *pasar por referencia*). Cuando un argumento se pasa mediante una llamada por valor, se hace una *copia* del valor del argumento y se pasa al método llamado.



### Tip para prevenir errores 25.6

*Con una llamada por valor, las modificaciones a la copia del método llamado no afecta el valor original de la variable en el método que llama. Esto evita los efectos colaterales accidentales que afectan grandemente el desarrollo de sistemas de software confiables.*

Con una llamada por referencia, quien realiza la llamada proporciona al método llamado la habilidad de acceder directamente a los datos de quien llama y de modificar esos datos, si el método llamado lo elige así. Las llamadas por referencia pueden mejorar el rendimiento, ya que eliminan la sobrecarga de copiar grandes cantidades de datos, pero pueden debilitar la seguridad, ya que el método llamado puede acceder a los datos de quien lo llamó.



### Observación de ingeniería de software 25.5

*A diferencia de otros lenguajes, Java no permite al programador elegir si pasa cada argumento por medio de una llamada por valor o por medio de una llamada por referencia. Las variables de tipos de datos primitivos siempre se pasan por valor. Los objetos no se pasan hacia métodos; en su lugar, se pasan hacia los métodos las referencias a objetos. Las referencias mismas también se pasan por valor. Cuando un método recibe una referencia a un objeto, el método puede manipular directamente al objeto.*



### Observación de ingeniería de software 25.6

*Cuando se devuelve información desde un método a través de una instrucción* **`return`**, *las variables de tipos de datos primitivos siempre se devuelven por valor (es decir, se devuelve una copia), y los objetos siempre se devuelven por referencia (es decir, se devuelve una referencia al objeto).*

Para pasar una referencia a un objeto hacia un método, simplemente especifique en la llamada al método el nombre de la referencia. Al mencionar la referencia por medio del nombre de su parámetro en el cuerpo del método llamado, en realidad se hace referencia al objeto original en memoria, y se puede acceder directamente al objeto original por medio del método llamado.

Java trata a los arreglos como objetos, por lo que éstos se pasan hacia los métodos por medio de una llamada por referencia; un método llamado puede acceder a los elementos de los arreglos originales de quien le llamó. El nombre de un arreglo en realidad es una referencia a un objeto que contiene los elementos del arreglo y la variable de instancia **`lenght`**, la cual indica el número de elementos en el arreglo. En la siguiente sección demostraremos las llamadas por valor y las llamadas por referencia, utilizando arreglos.

**Tip de rendimiento 25.3**

*Pasar arreglos por referencia tiene sentido por razones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Por mucho, pasar con frecuencia arreglos implicaría una pérdida de tiempo y de almacenamiento para las copias de los arreglos.*

## 25.12 Arreglos con múltiples subíndices

Los arreglos con múltiples subíndices (con dos subíndices) con frecuencia se utilizan para representar *tablas* de valores que consisten en información acomodada en *filas* y *columnas*. Para identificar un elemento en particular de una tabla, debemos especificar los dos subíndices; por convención, el primero identifica la fila del elemento, y el segundo identifica a la columna. Los arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como *arreglos con dos subíndices*. Observe que los arreglos con múltiples subíndices pueden tener más de dos subíndices. Java no soporta directamente los arreglos con múltiples subíndices, pero permite al programador especificar arreglos con un solo subíndice, cuyos elementos también son arreglos con un solo subíndice, con lo que se logra el mismo efecto. La figura 25.22 ilustra un arreglo con dos subíndices, **a**, que contiene tres filas y cuatro columnas (es decir, un arreglo de 3 por 4). En general, a un arreglo con *m* filas y *n* columnas se le conoce como arreglo de *m* por *n*.

Cada elemento del arreglo **a** se identifica en la figura 25.22, por medio de un nombre de elemento de la forma **a[i][j]**; **a** es el nombre del arreglo e **i** y **j** son los subíndices que identifican de manera única a la fila y a la columna de cada elemento en **a**. Observe que los nombres de los elementos en la primera fila tienen un primer subíndice **0**; los nombres de los elementos en la cuarta columna tienen un segundo subíndice **3**.

Los arreglos con múltiples subíndices pueden inicializarse en declaraciones como un arreglo con un solo subíndice. Un arreglo con dos subíndices **b[2][2]** podría declararse e inicializarse con

```
int b[][] = { { 1, 2 }, { 3, 4 } };
```

Los valores se agrupan por fila, entre llaves. Entonces, **1** y **2** inicializan **b[0][0]** y **b[0][1]**, y **3** y **4** inicializan **b[1][0]** y **b[1][1]**. El compilador determina el número de filas, contando el número de listas de subinicialización (representadas por conjuntos de llaves) en la lista principal de inicialización. El compilador determina el número de columnas en cada fila, contando el número de valores inicializadores en la lista de subinicialización para esa fila.

Los arreglos con múltiples subíndices se mantienen como arreglos de arreglos. La declaración

```
int b[][] = { { 1, 2 }, { 3, 4, 5 } };
```

crea un arreglo entero **b** con la fila **0** que contiene dos elementos (**1** y **2**), y la fila **1** que contiene tres elementos (**3**, **4** y **5**).

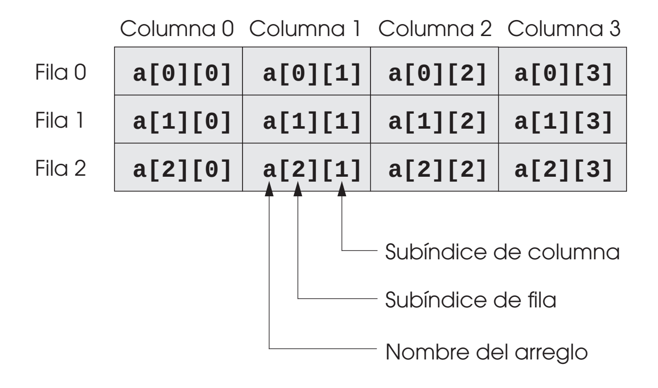



**Figura 25.22** Un arreglo con dos subíndices que consta de tres filas y cuatro columnas.

Un arreglo con múltiples subíndices con el mismo número de columnas en cada fila puede asignarse dinámicamente. Por ejemplo, un arreglo de 3 por 3 se asigna de la siguiente manera:

```
int b[][];
b = new int[ 3 ][ 3 ];
```

Así como con los arreglos de un solo subíndice, los elementos de un arreglo con dos subíndices se inicializan cuando **new** crea el objeto arreglo.

Un arreglo con múltiples subíndices en el que cada fila tiene un número diferente de columnas, puede asignarse dinámicamente de la siguiente manera:

```
int b[][];
b = new int[ 2 ][ ];        //asigna filas
b[ 0 ] = new int[ 5 ];      //asigna las columnas de la fila 0
b[ 1 ] = new int[ 3 ];      //asigna las columnas de la fila 1
```

El código anterior crea un arreglo bidimensional con dos filas. La fila **0** tiene cinco columnas y la fila **1** tiene tres.

El applet de la figura 25.23 muestra la inicialización de arreglos con dos subíndices en las declaraciones, y utiliza ciclos **for** anidados para recorrer los arreglos (es decir, para manipular cada elemento del arreglo).

```
// Figura 25.23: InicArreglo.java
// Inicialización de Arreglos multidimensionales
import java.awt.Container;
import javax.swing.*;

public class InicArreglo extends JApplet {
   JTextArea areaSalida;

   // inicializa el objeto
   public void init()
   {
      areaSalida = new JTextArea();
      Container c = getContentPane();
      c.add( areaSalida );

      int arreglo1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
      int arreglo2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };

      areaSalida.setText( “Los Valores en arreglo1 por fila son\n” );
      construyeSalida( arreglo1 );

      areaSalida.append( “\nLos Valores en arreglo2 por fila son\n” );
      construyeSalida( arreglo2 );
   } // fin del método init

   public void construyeSalida( int a[][] )
   {
      for ( int i = 0; i < a.length; i++ ) {

         for ( int j = 0; j < a[ i ].length; j++ )
            areaSalida.append( a[ i ][ j ] + “  “ );

         areaSalida.append( “\n” );
      } // fin de for
   } // fin del método construyeSalida
} // fin de la clase InicArreglo
```

**Figura 25.23** Inicialización de arreglos multidimensionales. (Parte 1 de 2.)

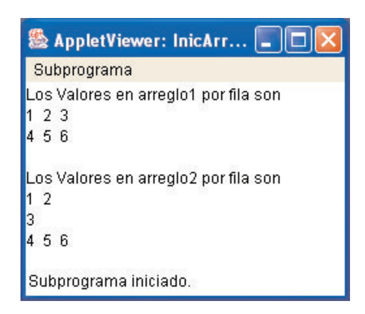



**Figura 25.23** Inicialización de arreglos multidimensionales. (Parte 2 de 2.)

El programa declara dos arreglos en el método **init**. La declaración de **arreglo1** (línea 16) proporciona seis inicializadores en dos sublistas. La primera sublista inicializa la primera fila del arreglo con los valores **1**, **2** y **3**; y la segunda sublista inicializa la segunda fila del arreglo con los valores **4**, **5** y **6**. La declaración de **arreglo2** (línea 17) proporciona seis inicializadores en tres sublistas. La sublista para la primera fila inicializa explícitamente la primera fila para que tenga dos elementos con los valores **1** y **2**, respectivamente. La sublista para la segunda fila inicializa la segunda fila para que tenga un elemento con el valor **3**. La sublista para la tercera fila inicializa la tercera fila con los valores **4**, **5** y **6**.

El método **init** llama al método **construyeSalida** de las líneas 20 y 23 para agregar los elementos de cada arreglo al **JTextArea areaSalida**. La definición del método **construyeSalida** especifica el parámetro del arreglo como **int a[][]** para indicar que un arreglo con dos subíndices se recibirá como argumento. Observe el uso de una estructura **for** anidada para desplegar las filas de cada arreglo con dos subíndices. En la estructura **for** externa, la expresión **a.lenght** determina el número de filas en el arreglo. En la estructura **for** interna, la expresión **a[i].lenght** determina el número de columnas en cada fila del arreglo. Esta condición permite al ciclo determinar, para cada fila, el número exacto de columnas.

Muchas manipulaciones comunes de arreglos utilizan estructuras de repetición **for**. Por ejemplo, la siguiente estructura **for** establece en cero a todos los elementos de la tercera fila del arreglo **a** correspondiente a la figura 25.22:

```
for ( int col = 0; col < a[ 2 ].lenght; col++ )
   a[ 2 ][ col ] = 0;
```

Nosotros especificamos la *tercera* fila, por lo tanto, sabemos que el primer subíndice siempre es **2** (**0** es la primera fila, y **1** es la segunda). El ciclo **for** varía sólo el segundo subíndice (es decir, el subíndice de columna). La estructura **for** anterior es equivalente a las instrucciones de asignación

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente estructura **for** anidada determina el total de todos los elementos del arreglo a:

```
int total = 0;

for ( int fila = 0; fila < a.lenght; fila++ )
   for ( int col = 0; col < a[ fila ].lenght; col++ )
      total += a[ fila ][ col ];
```

La estructura **for** totaliza los elementos del arreglo, una fila a la vez. La estructura **for** externa comienza estableciendo el subíndice de **fila** en **0**, por lo que los elementos de la primera fila pueden ser totalizados por la estructura **for** interna. La estructura **for** externa después incrementa en **1** a fila, por lo que la segunda fila puede ser totalizada. Después, la estructura **for** externa incrementa en **2** a fila, por lo que la tercera fila puede ser totalizada. El resultado puede desplegarse cuando la estructura **for** anidada termina.

## RESUMEN

- Los tipos primitivos (**`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** y **`double`**) son los bloques de construcción para tipos más complicados en Java.
- Java requiere que todas las variables tengan un tipo, antes de que puedan utilizarse en un programa. Por esta razón, Java se conoce como un lenguaje fuertemente basado en tipos.
- Los tipos primitivos en Java son portables a través de todas las plataformas de cómputo que soportan Java.
- Java utiliza estándares reconocidos internacionalmente para formatos de caracteres (Unicode) y de números de punto flotante (IEEE 754).
- A las variables de tipos **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** y **`double`** se les da el valor de **`0`**, de manera predeterminada, y a las de tipo **`boolean`** se les da el valor de **`false`**, también de manera predeterminada.
- Los operadores lógicos pueden utilizarse para formar condiciones complejas, combinando condiciones. Los operadores lógicos son **&&**, **&**, **`||`**, **`|`**, **`^`** y **`!`**, los cuales significan AND lógico, AND lógico booleano, OR lógico, OR lógico booleano incluyente, OR lógico booleano excluyente y NOT lógico (negación), respectivamente.
- La mejor forma de desarrollar y mantener un programa grande es dividirlo en diversos módulos de programas pequeños, los cuales son más manejables que el programa original. Los módulos se escriben en Java como clases y métodos.
- El área en donde se despliega un **`JApplet`** en la pantalla tiene un panel de contenido al que los componentes GUI deben adjuntarse para que puedan desplegarse en tiempo de ejecución. El panel de contenido es un objeto de la clase **`Container`** del paquete **`java.awt`**.
- El método **`getContentPane`** devuelve una referencia hacia el panel de contenido del applet.
- El formato general para la definición de un método es

  *tipo del valor de retorno nombre del método*( *lista de parámetros* )

  {

  *declaraciones e instrucciones*

  }

  El *tipo del valor de retorno* establece el tipo del valor devuelto hacia el método que realiza la llamada. Si un método no devuelve un valor, el *tipo del valor de retorno* es **`void`**. El *nombre del método* es cualquier identificador válido. La *lista de parámetros* es una lista separada por comas que contiene las declaraciones de las variables que se pasarán al método. Si un método no recibe valor alguno, la *lista de parámetros* está vacía. El cuerpo del método es el conjunto de *declaraciones e instrucciones* que constituyen el método.
- Una lista de parámetros vacía se especifica con paréntesis vacíos.
- Los argumentos pasados a un método deben coincidir en número, tipo y orden con los parámetros en la definición del método.
- Cuando un programa encuentra un método, el control se transfiere del punto de invocación hacia el método llamado, el método se ejecuta, y el control regresa a quien hizo la llamada.
- Un método llamado puede devolver el control hacia quien hizo la llamada, de tres formas. Si el método no devuelve un valor, el control se devuelve cuando se alcanza la llave derecha del final del método, o ejecutando la instrucción

  **`return;`**
- Si el método devuelve un valor, la instrucción

  **`return`** *expresion*;

  devuelve el valor de *expresion*.
- El método **`Math.random`** genera un valor **`double`** mayor o igual que 0.0, pero menor que 1.0.
- Los valores producidos por **`Math.random`** pueden escalarse y desplazarse para producir valores en un rango en particular.
- La ecuación general para escalar y desplazar un número aleatorio es

```
n = a + (int) ( Math.random() * b );
```

donde **`a`** es el valor de desplazamiento (el primer número del rango deseado de enteros consecutivos) y **`b`** es el factor de escalamiento (el ancho del rango deseado de enteros consecutivos).

- Una clase puede heredar atributos y comportamientos existentes (datos y métodos) de otra clase especificada a la derecha de la palabra reservada **extends** en la definición de la clase. Además, una clase puede implementar una o más interfaces. Una interfaz especifica uno o más comportamientos (es decir, métodos) que usted puede definir en la definición de una clase.
- La interfaz **ActionListener** especifica que esta clase debe definir un método con la primera línea

```
public void actionPerformed( ActionEvent e )
```

- La tarea del método **actionPerformed** es la de procesar una interacción de usuario con un componente GUI que genera un evento de acción. Este método es llamado automáticamente, en respuesta a la interacción del usuario. A este proceso se le conoce como manejo de eventos. El evento es la interacción de usuario (oprimir el botón). El manejador de eventos es el método **actionPerformed**, el cual es llamado automáticamente en respuesta al evento. A este estilo de programación se le conoce como programación manejada por eventos.
- La palabra reservada **final** se utiliza para declarar variables constantes. Las variables constantes deben inicializarse una vez antes de utilizarlas, y no pueden modificarse después. Las constantes variables también se conocen como constantes nombradas o variables de sólo lectura.
- Una **JLabel** contiene una cadena de caracteres a desplegar en la pantalla. Normalmente, una **JLabel** indica el propósito de otro elemento de la interfaz gráfica de usuario en la pantalla.
- Los **JTextFields** se utilizan para obtener información desde el teclado o para desplegar información en la pantalla.
- Cuando el usuario oprime un **JButton**, normalmente el programa responde realizando una tarea.
- El método **setLayout** de **Container** define el administrador de diseño para la interfaz de usuario del applet. Los administradores de diseño se proporcionan para acomodar los componentes GUI en un **Container**, para efectos de presentación. Los administradores de diseño proporcionan las capacidades básicas de diseño que determinan la posición y el tamaño de cada componente GUI adjunto al contenedor. Esto permite al programador concentrarse en la "apariencia visual" básica, y deja a los administradores de diseño el procesamiento de la mayoría de los detalles de diseño.
- **FlowLayout** es el administrador de diseño más básico. Los componentes GUI se colocan en un **Container** de izquierda a derecha, en el orden en el que se adjuntan al **Container** por medio del método **add**. Cuando se alcanza el borde del contenedor, los componentes continúan en la siguiente línea.
- Antes de que pueda procesarse cualquier evento, cada componente GUI debe saber cuál objeto del programa define el método de manipulación de eventos que será llamado cuando ocurra un evento. El método **addActionListener** se utiliza para indicarle a un **JButton** que otro objeto está escuchando los eventos de acción y define el método **actionPerformed**. A esto se le llama registro del manipulador de eventos con el componente GUI (también quisiéramos llamarlo la línea que *empieza a escuchar*, ya que el applet ahora está escuchando los eventos del botón). Para responder a un evento de acción, debemos definir una clase que implemente un **ActionListener** (esto requiere que la clase también defina un método **actionPerformed**) y debemos registrar el manipulador de eventos con el componente GUI.
- El método **showStatus** recibe un argumento **String** y lo despliega en la barra de estado del **appletviewer** o navegador.
- El **appletviewer** o el navegador llama una vez al método **init** de un **applet**, cuando éste se carga para su ejecución. Este método realiza la inicialización de un applet. El método **start** del applet es llamado después de que el método **init** completa su ejecución y cada vez que el usuario del navegador regresa al la página HTML en donde reside el applet (después de explorar otra página HTML).
- El método **paint** es llamado después de que el método **init** completa su ejecución y una vez que el método **start** ha iniciado su ejecución para dibujar en el applet. También se le llama automáticamente cada vez que el applet necesita repintarse.
- El método **stop** es llamado cuando el applet debe suspender su ejecución; normalmente cuando el usuario del navegador abandona la página HTML en donde el applet reside.
- El método **destroy** de un applet es llamado cuando el applet está siendo removido de la memoria; normalmente cuando el usuario del navegador abandona la sesión de navegación.
- El método **repaint** puede ser llamado en un applet para ocasionar una llamada fresca a **paint**. El método **repaint** invoca a otro método llamado **update**, y le pasa el objeto **Graphics**. El método **update** borra cualquier dibujo que se haya hecho previamente en el applet, después invoca al método **paint**, y le pasa el objeto **Graphics**.
- Cuando se declara un arreglo, el tipo del arreglo y los corchetes pueden combinarse al principio de la declaración, para indicar que todos los identificadores de la declaración representan arreglos, como en

```
double[] arreglo1, arreglo2;
```

- Los elementos de un arreglo pueden inicializarse por declaración (utilizando listas de inicializadores), por asignación y por entrada.
- Java evita las referencias a los elementos que se encuentren más allá de los límites de un arreglo.
- Para pasar un arreglo a un método, se pasa el nombre del arreglo. Para pasar un solo elemento de un arreglo a un método, simplemente pase el nombre del arreglo, seguido por el subíndice (contenido entre corchetes) del elemento en particular.
- Los arreglos pasan a los métodos por medio de una llamada por referencia; por lo tanto, los métodos llamados pueden modificar los valores de los elementos en los arreglos originales de quien hace la llamada. Los elementos de tipos de datos primitivos correspondientes a un arreglo son pasados a los métodos por medio de llamadas por valor.
- Los arreglos pueden utilizarse para representar tablas de valores que consisten en información ordenada en filas y columnas. Para identificar un elemento particular de la tabla, se especifican dos subíndices: el primero identifica la fila en la que se encuentra el elemento, y el segundo la columna. Las tablas o arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como arreglos con dos subíndices.
- Un arreglo con dos subíndices puede incializarse con una lista de inicializadores de la forma

  *tipoArreglo nombreArreglo*[][] = { { *fila1 sublista* }, { *fila2*, *sublista* }, ... };

- Para crear dinámicamente un arreglo con un número fijo de filas y columnas, utilice

  *tipoArreglo nombreArreglo*[][] = **new** tipoArreglo[ numFilas ] [ numColumnas ];

- Para pasar una fila de un arreglo con dos subíndices a un método que recibe un arreglo con un solo subíndice, simplemente pase el nombre del arreglo seguido por el subíndice de fila.

## TERMINOLOGÍA

**a[i]**
**a[i][j]**
AND lógico (**&&**)
AND lógico booleano (**&**)
API de Java (biblioteca de clases de Java)
argumento en una llamada a un método
arreglo
arreglo con dos subíndices
arreglo con múltiples subíndices
arreglo con un solo subíndice
arreglo de *m por n*
barra de desplazamiento
break
clase
clase **ActionEvent**
clase **FlowLayout**
clase **Font** de **java.awt**
clase **JButton** del paquete **javax.swing**
clase **JLabel** del paquete **javax.swing**
clase **JScrollPane**
clase **JTextArea**
clase **JTextField** del paquete **javax.swing**
conjunto de caracteres ISO Unicode
constante nombrada
corchetes **[]**
cuadro de desplazamiento
declaración de un método
declarar un arreglo
definición de un método
desplazamiento
devolver
divide y vencerás
**double**
duración
efectos colaterales
elemento cero
elemento de probabilidad
elemento de un arreglo
error de desplazamiento en uno
escalar
evaluación de cortocircuito
expresión de tipo mixto
final
**Font.BOLD**
**Font.ITALIC**
**Font.PLAIN**
formato tabular
generación de números aleatorios
ingeniería de software
inicialización de un arreglo
inicializador
interfaz **ActionListener**
invocar a un método
lista de inicialización de arreglos
llamada a un método
**long**
**Math.E**
**Math.PI**
método
método **actionPerformed**
método **append** de la clase **JTextArea**
método definido por el programador
método **destroy** de **JApplet**
método **init** de **JApplet**
método llamado
método **Math.random**
método **paint** de **JApplet**
método que llama
método **repaint** de **JApplet**
método **setFont**
método **setLayout** de **JApplet**
método **showStatus** de **JApplet**
método **start** de **JApplet**
método **stop** de **JApplet**
método **update** de **JApplet**
métodos de la clase **Math**
negación lógica (**!**)
nombre de un arreglo
operador **!**
operador **&&**
operador **||**
operador de llamada a un método, **()**

operador unario
operadores lógicos
OR lógico (**||**)
OR lógico bolleano incluyente (**|**)
OR lógico booleano excluyente (**^**)
parámetro de referencia
paso de arreglos a métodos
paso por referencia
paso por valor
programa modular
pulgar de una barra de desplazamiento
reutilización de software
simulación
sobrecarga de métodos
subíndice
subíndice de columna
subíndice de fila
tabla de valores
tipo de valor de retorno
tipos de referencias
valor de un elemento
verificación de límites
**void**

## ERRORES COMUNES DE PROGRAMACIÓN

**25.1** Utilizar una palabra reservada como identificador, es un error de sintaxis.

**25.2** En expresiones que utilizan el operador **&&**, es posible que una condición (a la que llamaremos condición dependiente) requiera de otra condición para ser **true**, de tal modo que ésta tenga sentido al evaluar la condición dependiente. En este caso, la condición dependiente debe colocarse después de la otra condición, o es posible que ocurra un error.

**25.3** Definir un método fuera de las llaves correspondientes a la definición de una clase, es un error de sintaxis.

**25.4** Omitir el tipo del valor de retorno en la definición de un método, es un error de sintaxis.

**25.5** Olvidar devolver un valor por parte de un método que se supone debe hacerlo, es un error de sintaxis. Si se especifica un tipo de valor de retorno diferente de **void**, el método debe contener una instrucción **return**.

**25.6** Devolver un valor desde un método, cuyo tipo de retorno se declaró como **void**, es un error de sintaxis.

**25.7** Declarar parámetros del mismo tipo en un método, como **float x**, **y**, en lugar de **float x**, **float y**, es un error de sintaxis, ya que se necesitan tipos para cada parámetro de la lista de parámetros.

**25.8** Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de una definición de método, es un error de sintaxis.

**25.9** Redefinir un parámetro de un método como una variable local del método, es un error de sintaxis.

**25.10** Pasar un método a un argumento que no es compatible con el tipo correspondiente al parámetro, es un error de sintaxis.

**25.11** Definir un método dentro de otro, es un error de sintaxis.

**25.12** Después de que se inicializó una variable **final**, intentar asignar otro valor a esa variable es un error de sintaxis.

**25.13** Proporcionar una definición para uno de los métodos de **JApplet init**, **start**, **paint**, **stop** o **destroy**, que no coincida con los encabezados de los métodos que muestra la figura 25.14, dará como resultado un método que no será llamado automáticamente durante la ejecución del applet.

**25.14** A diferencia de C o de C++, el número de elementos en el arreglo nunca se especifica entre corchetes después del nombre del arreglo en una declaración. La declaración **int c[12];** ocasiona un error de sintaxis.

**25.15** Asignar un valor a una constante variable después de inicializarla, es un error de sintaxis.

**25.16** Hacer referencia a un elemento que se encuentra fuera de los límites de un arreglo, es un error lógico.

## TIPS PARA PREVENIR ERRORES

**25.1** Los métodos pequeños son más fáciles de probar, depurar y comprender, que aquellos que son grandes.

**25.2** Cuando se ejecuta un programa en Java, el intérprete de Java verifica los subíndices de los elementos del arreglo para asegurarse de que son válidos (es decir, todos los subíndices deben ser mayores o iguales que 0, y menores que la longitud del arreglo). Si hay un subíndice inválido, Java genera una excepción.

**25.3** Las excepciones se utilizan para indicar que ocurrió un error en un programa. Éstas permiten que el programador se recupere del error y continúe con la ejecución del programa, en lugar de terminarlo de manera anormal. Cuando se hace una referencia inválida hacia un arreglo, se genera una excepción **ArrayIndexOutOfBoundsException**.

**25.4** Cuando se hace un ciclo a través de un arreglo, los subíndices de éste nunca deben ir por debajo de 0 y siempre deben ser menores que el número total de elementos del arreglo (uno menos que el tamaño de éste). Asegúrese de que la condición de terminación del ciclo evite el acceso a elementos fuera de este rango.

**25.5** Los programas deben validar que todos los valores de entrada sean correctos para prevenir que información errónea afecte a los cálculos de un programa.

**25.6** Con una llamada por valor, las modificaciones a la copia del método llamado no afecta el valor original de la variable en el método que llama. Esto evita los efectos colaterales accidentales que afectan grandemente el desarrollo de sistemas de software confiables.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**25.1** Por claridad, evite expresiones con efectos colaterales en las condiciones. Los efectos colaterales pueden parecer convenientes, pero con frecuencia representan más problemas que ventajas.

**25.2** Coloque una línea en blanco entre las definiciones de métodos para separarlos y para mejorar la legibilidad del programa.

**25.3** Aunque no es incorrecto hacerlo, en la definición de un método no utilice los mismos nombres para los argumentos pasados a él y para los parámetros correspondientes. Esto ayuda a evitar la ambigüedad.

**25.4** Elegir nombres descriptivos para los métodos y para los parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.

**25.5** Sólo utilice letras mayúsculas (con guiones bajos entre las palabras) en los nombres de variables **`final`**. Esto hace que las constantes resalten en un programa.

**25.6** Utilizar variables **`final`** con nombres descriptivos, en lugar de utilizar constantes enteras (como 2), hace que los programas sean legibles.

**25.7** Esfuércese en favor de la claridad de sus programas. Algunas veces vale la pena sacrificar el uso más eficiente de la memoria o el tiempo de procesamiento, a favor de escribir programas más claros.

## TIPS DE RENDIMIENTO

**25.1** En expresiones que utilizan el operador **&&**, si las condiciones separadas son independientes una de la otra, haga que la condición que más probablemente sea falsa, se encuentre más a la izquierda. En expresiones que utilizan el operador **`||`**, haga que la condición que más probablemente sea verdadera, se encuentre más a la izquierda. Esto puede reducir el tiempo de ejecución de un programa.

**25.2** Algunas veces las consideraciones de rendimiento se contraponen a las consideraciones de claridad.

**25.3** Pasar arreglos por referencia tiene sentido por razones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Por mucho, pasar con frecuencia arreglos implicaría una pérdida de tiempo y de almacenamiento para las copias de los arreglos.

## TIP DE PORTABILIDAD

**25.1** Todos los tipos de datos primitivos en Java son portables, a través de todas las plataformas que soportan Java.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**25.1** Por lo general, un método no debe sobrepasar una página. Mejor aún, un método generalmente debe abarcar no más de media página. Independientemente de cuán largo sea un método, debe realizar bien una tarea. Los métodos pequeños promueven la reutilización de software.

**25.2** Los programas deben escribirse como colecciones de métodos pequeños. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.

**25.3** Es posible que un método que requiere un gran número de parámetros esté realizando demasiadas tareas. Considere dividir el método en métodos más pequeños que realicen tareas separadas. Si es posible, el encabezado del método debe caber en una línea.

**25.4** El encabezado de un método y las llamadas a él deben coincidir en número, tipo y orden de parámetros y argumentos.

**25.5** A diferencia de otros lenguajes, Java no permite al programador elegir si pasa cada argumento por medio de una llamada por valor o por medio de una llamada por referencia. Las variables de tipos de datos primitivos siempre se pasan por valor. Los objetos no se pasan hacia métodos; en su lugar, se pasan hacia los métodos las referencias a

objetos. Las referencias mismas también se pasan por valor. Cuando un método recibe una referencia a un objeto, el método puede manipular directamente al objeto.

**25.6** Cuando se devuelve información desde un método a través de una instrucción **`return`**, las variables de tipos de datos primitivos siempre se devuelven por valor (es decir, se devuelve una copia), y los objetos siempre se devuelven por referencia (es decir, se devuelve una referencia al objeto).

## *EJERCICIOS DE AUTOEVALUACIÓN*

**25.1** Complete los espacios en blanco:

a) A los módulos de un programa en Java se les conoce como ______ y ______.
b) A un método se le invoca con una ______.
c) A una variable conocida sólo dentro del método en el que está definida se le conoce como ______.
d) La instrucción ______ en un método llamado puede utilizarse para pasar el valor de una expresión de regreso hacia el método que hizo la llamada.
e) La palabra reservada ______ se utiliza en el encabezado de un método para indicar que el método no devuelve valor alguno.
f) Las tres formas de devolver el control desde un método llamada hasta el método que hizo la llamada son ______, ______ y ______.
g) El método ______ se invoca una vez, cuando un applet comienza su ejecución.
h) El método ______ se utiliza para producir números aleatorios.
i) El método ______ se invoca cada vez que el usuario de un navegador vuelve a visitar la página HTML en la que reside el applet.
j) El método ______ se invoca para dibujar en un applet.
k) El método ______ invoca al método update del applet, el cual a su vez invoca al método paint del applet.
l) El método ______ se invoca para un applet cada vez que el usuario de un navegador abandona la página HTML en la que reside el applet.
m) El calificador ______ se utiliza para declarar variables de sólo lectura.

**25.2** Proporcione el encabezado del método para cada uno de los siguientes:

a) Método **`hipotenusa`**, el cual toma dos argumentos de punto flotante de doble precisión, **`lado1`** y **`lado2`**, y devuelve un resultado de punto flotante de doble precisión.
b) Método **`masPequenio`**, el cual toma tres enteros, **`x`**, **`y`**, **`z`**, y devuelve un entero.
c) Método **`instrucciones`**, el cual no toma argumentos y no devuelve valor alguno. [*Nota:* Tales métodos normalmente se utilizan para desplegar instrucciones para el usuario.]
d) Método **`intToFloat`**, el cual toma un argumento entero, **`numero`**, y devuelve un resultado de punto flotante.

**25.3** Encuentre el error en cada uno de los siguientes segmentos de programa, y explique cómo corregirlo:

a)
```
int g() {
    System.out.println( "Dentro del metodo g" );
    int h() {
       System.out.println( "Dentro del metodo h" );
    }
}
```
b)
```
int suma( int x, int y ) {
    int resultado ;
    resultado = x + y ;
}
```
c)
```
int suma( int n ) {
    if( n == 0 )
       return 0 ;
    else
       n + suma( n – 1 ) ;
}
```
d)
```
void f( float a ) ; {
    float a ;
    System.out.println( a ) ;
}
```

e)
```
void product() {
    int a = 6, b = 5, c = 4, resultado ;
    resultado = a * b * c;
    System.out.println( "El resultado es " + resultado );
    return resultado;
}
```

**25.4** Establezca si los siguientes enunciados son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.

a) Un arreglo puede almacenar muchos tipos diferentes de valores.
b) Un subíndice de arreglo normalmente debe ser del tipo de dato float.
c) Un elemento individual de un arreglo que se pasa a un método y se modifica en ese método contendrá el valor modificado cuando el método llamado complete su ejecución.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**25.1** a) Métodos y clases. b) Llamada a un método. c) Variable local. d) **return**. e) **void**. f) **return**; o **return** *expresion*; o al encontrar la llave derecha de cierre de método. g) **init**. h) **Math.random**. i) **start**. j) **paint**. k) **repaint**. l) **stop**. m) **final**.

**25.2** a) **double hipotenusa( double lado1, double lado2 )**
b) **int masPequenio( int x, int y, int z )**
c) **void instrucciones()**
d) **float intToFloat( int numero )**

**25.3** a) Error: el método h está definido en el método g.
Corrección: mueva la definición de h fuera de la definición de g.

b) Error: se supone que el método debe devolver un entero, pero no lo hace.
Corrección: elimine la variable **resultado** y coloque la siguiente instrucción en el método:

```
return x + y;
```

o agregue la siguiente instrucción al final del cuerpo del método:

```
return resultado;
```

c) Error: el resultado de **n + suma( n – 1 )** no es devuelto por este método recursivo, lo que ocasiona un error de sintaxis.
Corrección: rescriba la instrucción en la cláusula **else** como **return n + suma( n – 1);**

d) Error: el punto y coma después del paréntesis derecho que encierra la lista de parámetros, y definir el parámetro **a** en la definición del método es incorrecto.
Corrección: elimine el punto y coma después del paréntesis derecho de la lista de parámetros, y elimine la declaración **float a;**.

e) Error: el método devuelve un valor, cuando se supone que no debe hacerlo.
Corrección: cambie el tipo de retorno a **int**.

**25.4** a) Falso. Un arreglo puede almacenar solamente valores del mismo tipo.
b) Falso. Un subíndice de arreglo debe ser un entero o una expresión entera.
c) Falso. Para elementos individuales de tipos primitivos de un arreglo, ya que éstos son pasados mediante una llamada por valor. Si se pasa una referencia a un arreglo, entonces las modificaciones a los elementos del arreglo se reflejan en el original. Además, un elemento individual de un tipo de clase pasado a un método, se pasa por medio de una llamada por referencia, y las modificaciones al objeto se reflejarán en el elemento original del arreglo.

## EJERCICIOS

**25.5** Responda cada una de las siguientes preguntas:

a) ¿Qué significa elegir números "al azar"?
b) ¿Por qué el método **Math.random** es útil para simular juegos de azar?
c) ¿Por qué con frecuencia es necesario escalar y/o desplazar los valores producidos por **Math.random**?
d) ¿Por qué la simulación computarizada de situaciones reales es una técnica útil?

**25.6** Escriba instrucciones que asignen enteros aleatorios a la variable **n** en los siguientes rangos:
a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

**25.7** Para cada uno de los siguientes conjuntos de enteros, escriba una sola instrucción que imprima un número al azar del conjunto.
a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

**25.8** Defina un método **hipotenusa** que calcule la longitud de la hipotenusa de un triángulo recto, cuando se conocen los otros dos lados. El método debe tomar dos argumentos de tipo **double** y debe devolver la hipotenusa como un **double**. Incorpore este método a un applet que lea valores enteros para **lado1** y **lado2** desde **JTextFields**, y que realice el cálculo con el método **hipotenusa**. Determine la longitud de la hipotenusa para cada uno de los siguientes triángulos. [*Nota:* Registre la manipulación de eventos sólo en el segundo **JTextField**. El usuario debe interactuar con el programa, escribiendo los números en ambos **JTextFields** y oprimir *Entrar* en el segundo **JTextField**.]

| Triángulo | Lado 1 | Lado 2 |
|---|---|---|
| 1 | 3.0 | 4.0 |
| 2 | 5.0 | 12.0 |
| 3 | 8.0 | 15.0 |

**25.9** Escriba un método **multiplo** que determine para un par de enteros, si el segundo es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver **true** si el segundo es múltiplo del primero; de lo contrario, debe devolver **false**. Incorpore este método a un applet que introduzca una serie de pares de enteros (un par a la vez, utilizando **JTextFields**). [*Nota:* Registre la manipulación de eventos sólo en el segundo **JTextField**. El usuario debe interactuar con el programa, escribiendo los números en ambos **JTextFields** y oprimir *Entrar* en el segundo **JTextField**.]

**25.10** Escriba un applet que introduzca enteros (uno a la vez) y que los pase, uno a la vez, hacia el método **esPar**, el cual utiliza el operador módulo para determinar su un entero es par. El método debe tomar un argumento entero y devolver **true** si el entero es par y **false** de lo contrario. Utilice un diálogo de entrada para obtener los datos del usuario.

**25.11** Escriba un método **cuadradoDeAsteriscos** que despliegue un cuadrado sólido de asteriscos, cuyo lado se especifica en el parámetro entero **lado**. Por ejemplo, si **lado** es **4**, el método despliega

```
****
****
****
****
```

Incorpore este método en un applet que lea un valor entero para **lado**, proporcionado desde el teclado, y que realice el dibujo por medio del método **cuadradoDeAsteriscos**. Observe que este método debe ser llamado desde el método **paint** del applet, y el objeto **Graphics** debe ser pasado desde **paint**.

**25.12** Implemente los siguientes métodos de enteros:
a) El método **celsius** devuelve el equivalente Celsius de la temperatura en Fahrenheit, por medio del cálculo

```
C = 5.0 / 9.0 * (F – 32 );
```

b) El método **fahrenheit** devuelve el equivalente Fahrenheit de la temperatura en Celsius.

```
F = 9.0 / 5.0 * C + 32;
```

c) Utilice estos métodos para escribir un applet que permita al usuario introducir una temperatura en Fahrenheit y que se despliegue en Celsius, o que introduzca una temperatura en Celsius y que se despliegue en Fahrenheit.

[*Nota:* Este applet requerirá dos objetos **`JTextField`** que hayan registrado eventos de acción. Cuando se invoca a **`actionPerformed`**, el parámetro **`ActionEvent`** tiene al método **`getSource()`** para determinar el componente GUI con el que el usuario interactuó. Su método **`actionPerformed`** debe contener una estructura **`if/else`** de la siguiente forma:

```
if ( e.getSource() == entrada1 ) {
   // procesa la interacción entrada1 aquí
}
else { // e.getSource() == entrada2
   // procesa la interacción entrada2 aquí
}
```

donde **`entrada1`** y **`entrada2`** son referencias de **`JTextField`**.

**25.13** Se dice que un entero es *primo* si sólo es divisible entre 1 y entre él mismo. Por ejemplo, 2, 3, 5 y 7 son números primos, pero 4, 6, 8 y 9, no lo son.

a) Escriba un método que determine si un número es primo.

b) Utilice este método en un applet que determine e imprima todos los números primos entre 1 y 10,000. ¿Cuántos de estos 10,000 números realmente tiene que evaluar, antes de estar seguro de que encontró a todos los primos? Despliegue el resultado en un **`JTextArea`** que tenga la funcionalidad de desplazamiento.

c) De entrada, usted podría pensar que $n/2$ es el límite superior de los números que tiene que evaluar para ver si un número es primo, pero sólo necesita ir hasta la raíz cuadrada de $n$. ¿Por qué? Rescriba el programa y ejecútelo en ambas formas. Calcule un estimado de la mejoría en el rendimiento.

**25.14** Escriba un método que tome un valor entero y que devuelva el número con sus dígitos a la inversa. Por ejemplo, dado el número 7631, el método debe devolver 1367. Incorpore el método en un applet que lea un valor del usuario. Despliegue el resultado del método en la barra de estado.

**25.15** El *máximo común divisor (MCD)* de dos enteros es el entero más grande que divide en partes iguales a cada uno de los dos números. Escriba un método **`mcd`** que devuelva el máximo común divisor de dos enteros. Incorpore el método en un applet que lea dos valores del usuario. Despliegue el resultado del método en la barra de estado.

**25.16** Escriba un método **`puntosCalidad`** que introduzca el promedio de un estudiante, y que devuelve 4 si el promedio de un estudiante es 90 a 100, 3 si el promedio es de 80 a 89, 2 si el promedio es de 70 a 79, 1 si el promedio es de 60 a 69 y 0 si el promedio es menor que 60. Incorpore el método en un applet que lea un valor del usuario. Despliegue el resultado del método en la barra de estado.

**25.17** Escriba un applet que simule el lanzamiento de una moneda. Deje que el programa lance la moneda, cada vez que el usuario oprima el botón "**`Lanzar`**". Cuente el número de veces que aparece cada lado de la moneda. Despliegue los resultados. El programa debe llamar a un método separado **`tirar`** que no tome argumentos y que devuelva **`false`** para la cruz y **`true`** para la cara. [*Nota:* Si el programa simula en forma realista el lanzamiento de la moneda, cada lado de la moneda debe aparecer aproximadamente en la mitad de las ocasiones que ésta se lance.]

**25.18** Las computadoras tienen un papel cada vez más importante en la educación. Escriba un programa que ayude a un estudiante de primaria a aprender a multiplicar. Utilice **`Math.random`** para producir dos enteros positivos de un dígito. El programa debe desplegar después una pregunta en la barra de estado como

```
¿cuánto es 6 por 7?
```

El estudiante después debe escribir la respuesta en un **`JTextField`**. Su programa verifica la respuesta del estudiante. Si es correcta, dibuje la cadena **`"Muy bien!"`** en el applet, después haga otra pregunta. Si la respuesta es incorrecta, dibuje la cadena **`"No. Pruebe otra vez."`** en el applet, después espere a que el estudiante intente otra vez repetidamente hasta que finalmente dé la respuesta correcta. Debe utilizar un método separado para generar cada nueva pregunta. Este método debe ser llamado una vez que el applet comience su ejecución, y cada vez que el usuario responda correctamente. Todos los dibujos del applet deben realizarse por medio del método **`paint`**.

**25.19** Escriba un applet que juegue a "adivinar el número" de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero al azar en el rango 1 a 1000. El applet despliega la indicación **`Adivine un numero entre 1 y 1000`** junto a un **`JTextField`**. El jugador escribe un primer intento en el **`JTextField`** y oprime la tecla *Entrar*. Si el jugador no adivinó, su programa debe desplegar **`Demasiado alto. Intente otra vez`**, o **`Demasiado bajo. Intente otra vez`** en la barra de estado, para ayudar al jugador a "concentrarse" en la respuesta correcta, y debe limpiar el **`JTextField`** para que el usuario pueda introducir el siguiente intento.

Cuando el usuario introduzca la respuesta correcta, despliegue **`Felicidades. Adivino el numero!`** en la barra de estado, y limpie el **`JTextField`** para que el usuario pueda jugar de nuevo. [*Nota:* La técnica para adivinar que empleamos en este problema es parecida a la de la *búsqueda binaria*.]

**25.20** El máximo común divisor de los enteros **`x`** y **`y`** es el entero más grande que divida en partes iguales tanto a **`x`** como a **`y`**. Escriba un método recursivo **`mcd`** que devuelva el máximo común divisor de **`x`** y **`y`**. El **`mcd`** de **`x`** y **`y`** se define recursivamente de la siguiente forma: si **`y`** es igual que 0, entonces el **`mcd( x, y )`** es **`x`**; de lo contrario, **`mcd( x, y )`** es **`mcd( y, x%y )`**, donde **`%`** es el operador módulo. Utilice este método para reemplazar el que escribió en el applet del ejercicio 25.15.

**25.21** Modifique el programa de craps de la figura 25.13 para permitir las apuestas. Inicialice en 1000 dólares la variable **`saldoBanco`**. Indique al usuario que introduzca una **`apuesta`**. Verifique que la **`apuesta`** sea menor o igual que **`saldoBanco`**, y si no es así, haga que el usuario reintroduzca una apuesta, hasta que introduzca una válida. Después de que introduzca una **`apuesta`** correcta, ejecute un juego de craps. Si el jugador gana, incremente **`saldoBanco`** en el monto de la **`apuesta`** e imprima el nuevo **`saldoBanco`**. Si el jugador pierde, disminuya **`saldoBanco`** en el monto de la **`apuesta`**, imprima el nuevo **`saldoBanco`**, verifique si éste se ha vuelto cero, y si es así, imprima el mensaje **`“Lo siento. Se quedó sin un centavo!”`** Conforme progrese el juego, imprima varios mensajes para generar cierto “cotorreo”, como **`“Oh, va directo a la quiebra”`**, o **`“Ande, atrévase!”`**, o **`“Está ganando. Ahora es el momento de capitalizar!”`**. Implemente el “cotorreo” como un método separado que elija al azar la cadena a desplegar.

**25.22** Escriba un programa para simular el tiro de dos dados. El programa debe utilizar **`Math.random`** para tirar el primer dado, y debe utilizar **`Math.random`** nuevamente para tirar el segundo dado. La suma de los dos valores debe entonces calcularse. [*Nota:* Debido a que cada dado puede mostrar un valor entero entre 1 y 6, la suma de los valores variará de 2 a 12, en donde 7 es la suma más frecuente, y 2 y 12 son las sumas menos frecuentes. La figura 25.24 muestra las 36 posibles combinaciones de los dos dados. Su programa debe tirar el dado 36,000 veces. Utilice un arreglo con un solo subíndice para que lleve la cuenta del número de veces que cada posible suma aparece. Imprima los resultados en un formato tabular. Además, determine si los totales son razonables (es decir, existen seis formas de tirar un 7, por lo que aproximadamente un sexto de todos los tiros debe resultar en 7).]

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Figura 25.24** Las 36 posibles salidas del tiro de dos dados.

# 26

# Programación orientada a objetos con Java




## Objetivos

- Comprender el encapsulamiento y el ocultamiento de información.
- Comprender los fundamentos de la abstracción de datos y los tipos de datos abstractos (ADTs).
- Crear ADTs en Java, a saber, clases.
- Crear, utilizar y destruir objetos.
- Controlar el acceso a las variables de instancia de objetos y a los métodos.
- Apreciar el valor de la orientación a objetos.
- Comprender el uso de la referencia **`this`**.
- Comprender las variables de clase y los métodos de clase.

*Mi objetivo completamente sublime*
*Deberé lograrlo a tiempo.*
W. S. Gilbert

*¿Es éste un mundo en el que se deben esconder las virtudes?*
William Shakespeare

*Tus sirvientes públicos te sirven bien.*
Adlai Stevenson

*Pero acaso, para servir a nuestros fines personales,*
*¿Perdonamos el engaño a nuestros amigos?*
Charles Churchill

*Por sobre todas las cosas: sé auténtico.*
William Shakespeare

*No tengas amigos diferentes a ti mismo.*
Confucio

## Plan general



## 26.1 Introducción

Ahora estudiaremos la orientación a objeto en Java. Si usted ya leyó la introducción a la orientación a objetos en C++ (capítulo 16) podría saltar directo a la sección 26.2, en donde echamos un vistazo por primera vez a una implementación orientada a objetos en Java.

Revisemos brevemente algunos conceptos clave y la terminología de la orientación a objetos. La programación orientada a objetos (POO) *encapsula* datos (*atributos*) y métodos (*comportamientos*) dentro de *objetos*; los datos y los métodos de un objeto se encuentran íntimamente ligados entre sí. Los objetos tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos pueden saber cómo comunicarse entre sí, a través de *interfaces* bien definidas, por lo general a los objetos no se les permite saber cómo se implementan otros objetos; los detalles de implementación están ocultos dentro de los mismos objetos. Con toda seguridad es posible conducir un automóvil de manera efectiva sin conocer los detalles de cómo funcionan internamente los sistemas del motor, la transmisión y el escape. Veremos por qué el ocultamiento de información es tan importante para la buena ingeniería de software

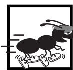
En C y en otros *lenguajes de programación por procedimientos*, la programación tiende a ser *orientada a acciones*. En Java, la programación es *orientada a objetos*. En C, la unidad de programación es la *función* (las cuales se conocen como *métodos* en Java). En Java, la unidad de programación es la *clase*, a partir de la cual se generan las *instancias* de todos los objetos (es decir, se crean). Las funciones no desaparecen en Java; en lugar de eso se encapsulan como métodos con los datos que procesan dentro de las "paredes" de las clases.

Los programadores en C se concentran en escribir funciones. Los conjuntos de acciones que realizan alguna tarea se agrupan en funciones, y las funciones se agrupan para formar programas. Los datos son importantes en C, pero la idea es que los datos existen primordialmente para apoyar las acciones que realizan las funciones. En la especificación de un sistema, los *verbos* ayudan al programador en C a determinar el conjunto de funciones que trabajarán juntas para implementar el sistema.

Los programadores en Java se concentran en crear sus propios *tipos definidos por el usuario* llamados *clases*. A las clases también se les denomina *tipos definidos por el programador*. Toda clase contiene datos, así como el conjunto de métodos que manipulan estos datos. A los datos que componen una clase se les llama *variables de instancia* (o *datos miembro*, en C++). Así como a una instancia de un tipo de dato predefinido como **`int`** se le llama *variable*, a una instancia de un tipo de dato definido por el usuario (es decir, a una clase) se le llama *objeto*. El foco de atención en Java se centra en los objetos, en lugar de en los métodos. Los *sustantivos* que se encuentran en las especificaciones de un sistema ayudan al programador en Java a determinar el conjunto de clases que utilizará para comenzar el proceso de diseño. Después, se utilizan las clases para crear las instancias de los objetos que trabajarán juntos para implementar un sistema.

Este capítulo explica cómo crear objetos, un tema al que nos gusta llamar *programación basada en objetos* (PBO). En el capítulo 27 introducimos la *herencia* y el polimorfismo, dos tecnologías clave que permiten

la verdadera *programación orientada a objetos (POO)*. Aunque no explicaremos con detalle la herencia hasta el capítulo 27, ésta es parte de toda definición de una clase en Java.



**Tip de rendimiento 26.1**

*Todos los objetos en Java se pasan por referencia. Sólo se pasa la dirección de memoria, no una copia de todo el objeto (como se haría en un paso por valor).*



**Observación de ingeniería de software 26.1**

*Es importante escribir programas que sean claros y fáciles de mantener. La regla es el cambio, en lugar de la excepción. Los programadores deben prever que su código será modificado. Como veremos pronto, las clases facilitan la modificación de un programa.*

## 26.2 Implementación del tipo de dato abstracto Hora con una clase

La aplicación de la figura 26.1 consta de dos clases, **`Hora1`** y **`PruebaHora`**. La clase **`Hora1`** se define en el archivo **`Hora1.java`** (especificado en la línea de comentario 1) y la clase **`PruebaHora`** se define en el archivo **`PruebaHora.java`** (especificada en la línea de comentario 49). [*Nota:* Todos los programas de este libro que contienen más de un archivo, comienzan con un comentario que indica el número de la figura y el nombre del archivo.] Aunque estas dos clases están definidas en archivos separados, numeramos consecutivamente las líneas del programa a lo largo de ambos archivos, por motivos de explicación en el texto. Es importante observar que estas clases *deben* definirse en archivos separados.

**Observación de ingeniería de software 26.2**

*Las definiciones de las clases que comienzan con la palabra reservada* **`public`** *deben almacenarse en un archivo que tiene el mismo nombre que la clase, y terminar con la extensión de archivo* **`.java`**.

**Error común de programación 26.1**

*Definir más de una clase pública en el mismo archivo, es un error de sintaxis.*

```
1  // Figura 26.1: Hora1.java
2  // Definición de la clase Hora1
3  import java.text.DecimalFormat;  // se utiliza para dar formato al número
4  
5  // Esta clase mantiene la hora en formato de 24 horas
6  public class Hora1 extends Object {
7     private int hora;     // 0 - 23
8     private int minuto;   // 0 - 59
9     private int segundo;  // 0 - 59
10 
11    // El constructor Hora1 inicializa en cero cada variable
12    // de instancia. Garantiza que cada vez que inicia el objeto Hora1
13    // lo hace en un estado consistente.
14    public Hora1()
15    {
16       estableceHora( 0, 0, 0 );
17    } // fin del constructor Hora1
18 
19    // Establece un nuevo valor de hora por medio del horario universal.
20    // Realiza validaciones a los datos. Establece en cero a los valores
         inválidos.
21    public void estableceHora( int h, int m, int s )
22    {
23       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
24       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
```

**Figura 26.1** Implementación del tipo de dato abstracto **`Hora1`** como una clase; **`Hora1.java`**. (Parte 1 de 2.)

```
         segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
      } // fin del método estableceHora

      // Convierte a String en formato de horario universal
      public String aCadenaUniversal()
      {
         DecimalFormat dosDigitos = new DecimalFormat( “00” );

         return dosDigitos.format( hora ) + “:” +
                dosDigitos.format( minuto ) + “:” +
                dosDigitos.format( segundo );
      } // fin del método aCadenaUniversal

      // Convierte a String en formato de horario estándar
      public String toString()
      {
         DecimalFormat dosDigitos = new DecimalFormat( “00” );

         return ( ( hora == 12 || hora == 0 ) ? 12 : hora % 12 ) +
                “:” + dosDigitos.format( minuto ) +
                “:” + dosDigitos.format( segundo ) +
                ( hora < 12 ? “ AM” : “ PM” );
      } // fin del método toString
   } // fin de la clase Hora1
```

**Figura 26.1** Implementación del tipo de dato abstracto **Hora1** como una clase; **Hora1.java**. (Parte 2 de 2.)

```
   // Figura 26.1: PruebaHora.java
   // Clase PruebaHora para ejercitar la clase Hora1
   import javax.swing.JOptionPane;

   public class PruebaHora {
      public static void main( String args[] )
      {
         Hora1 h = new Hora1();  // llama al constructor Hora1
         String salida;

         salida = “La hora universal inicial es: “ +
                  h.aCadenaUniversal() +
                  “\nLa hora estandar inicial es: “ +
                  h.toString() +
                  “\nLlamada implicita a toString(): “ + h;

         h.estableceHora( 13, 27, 6 );
         salida += “\n\nLa hora universal despues de estableceHora es: “ +
                   h.aCadenaUniversal() +
                   “\nLa hora estandar despues de estableceHora es: “ +
                   h.toString();

         h.estableceHora( 99, 99, 99 );  // todos son valores inválidos
         salida += “\n\nDespues de intentar establecer valores invalidos: “ +
                   “\nHora universal: “ + h.aCadenaUniversal() +
```

**Figura 26.1** Implementación del tipo de dato abstracto **Hora1** como una clase; **PruebaHora.java**. (Parte 1 de 2.)

```
74                     "\nHora estandar: " + h.toString();
75
76          JOptionPane.showMessageDialog( null, salida,
77             "Probando la clase Hora1",
78             JOptionPane.INFORMATION_MESSAGE );
79
80          System.exit( 0 );
81       } // fin de main
82    } // fin de la clase PruebaHora
```

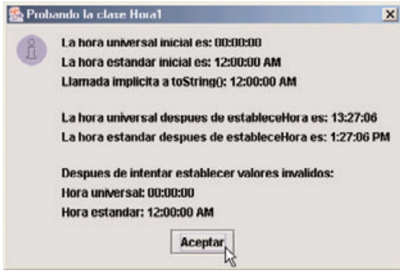



**Figura 26.1** Implementación del tipo de dato abstracto **`Hora1`** como una clase; **`PruebaHora.java`**. (Parte 2 de 2.)

La figura 26.1 (líneas 1 a 48) contienen una sencilla definición para la clase **`Hora1`**. Nuestra definición de la clase **`Hora1`** comienza en la línea 6

```
public class Hora1 extends Object {
```

la cual indica que la clase **`Hora1`** **`extiende`** a la clase ***`Object`*** (del paquete **`java.lang`**). Recuerde que usted realmente no crea una definición de clase "desde cero". De hecho, cuando crea una definición de clase, siempre utiliza piezas de definiciones de clase existentes. Java utiliza la *herencia* para crear nuevas clases a partir de definiciones de clases existentes. La palabra reservada **`extends`** seguida por el nombre de clase **`Object`** indica la clase (en este caso **`Hora1`**) a partir de la cual nuestras nuevas clases heredan sus piezas existentes. En esta relación de herencia, a **`Object`** se le llama *superclase* o *clase base* y a **`Hora1`** se le llama *subclase* o *clase derivada*. El uso de la herencia da como resultado una nueva definición de clase que tiene *atributos* (datos) y *comportamientos* (métodos) de la clase **`Object`**, así como las nuevas características que agregamos a nuestra definición de la clase **`Hora1`**. Toda clase en Java es una subclase de **`Object`**. Por lo tanto, cada clase hereda los once métodos definidos por la clase **`Object`**. Un método clave de **`Object`** es ***`toString`***, el cual explicaremos más adelante en esta sección. Explicaremos otros métodos de la clase **`Object`** a través del libro, cuando sea necesario.



### Observación de ingeniería de software 26.3

*Toda clase definida en Java debe ser una extensión de otra clase. Si la clase no utiliza explícitamente la palabra reservada* **`extends`** *en su definición, esta clase implícitamente se extiende de* **`Objects`**.

El *cuerpo* de la definición de una clase se delinea con una llave izquierda y una llave derecha ( **`{`** y **`}`** ) en las líneas 6 a 48. La clase **`Hora1`** contiene tres variables de instancia enteras, **`hora`**, **`minuto`** y **`segundo`**, que representan el tiempo en formato de *horario universal* (*formato de reloj de 24 horas*).

Las palabras reservadas ***`public`*** y **`private`** son *modificadores de acceso a miembros*. Las variables de instancia o métodos que se declaran con el modificador de acceso a miembros **`public`** son accesibles desde cualquier punto en donde el programa haga referencia al objeto **`Hora1`**. Las variables de instancia o métodos que se declaran con el modificador de acceso a miembros **`private`** *solamente* están accesibles para los métodos de la clase. A cada variable de instancia o definición método le debe anteceder un modificador de acceso a miembros. Los modificadores de acceso a miembros pueden aparecer varias veces en cualquier orden en una definición de clase.



### Buena práctica de programación 26.1

*Agrupe los miembros de acuerdo con los modificadores de acceso a miembros dentro de la definición de una clase, para mayor claridad y legibilidad.*



### Error común de programación 26.2

*El hecho de que un método que no es un miembro de una clase en particular intente acceder a un miembro privado de dicha clase, es un error de sintaxis.*

Las tres variables de instancia enteras **`hora`**, **`minuto`** y **`segundo`** se declaran (líneas 7 a 9) con el modificador de acceso a miembros **`private`**. Esto indica que las variables de instancia de la clase son las únicas accesibles para los métodos de la clase. Cuando se crea la instancia de un objeto de la clase, dichas variables de instancia se encapsulan dentro del objeto y se puede acceder a ellas solamente a través de los métodos del objeto de la clase (por lo general, a través de los métodos públicos de la clase). Las variables de instancia normalmente se declaran como **`private`**, y los métodos por lo general se declaran como **`public`**. Es posible tener métodos privados y datos públicos, como veremos más adelante. A los métodos privados a menudo se les llama *métodos de utilidad* o *métodos de ayuda* debido a que solamente se les puede llamar mediante otros métodos de la clase, y se utilizan para soportar la operación de dichos métodos. El uso de datos públicos no es común y es una práctica peligrosa de programación.



### Buena práctica de programación 26.2

*Nosotros preferimos listar primero a las variables de instancia **`private`** de una clase, para que conforme lea el código, vea los nombres y los tipos de dichas variables, antes de utilizarlas en los métodos de la clase.*



### Buena práctica de programación 26.3

*A pesar del hecho de que los miembros públicos y privados pueden repetirse y mezclarse, primero liste en un grupo a todos los miembros privados de la clase, y después liste en otro grupo a todos los miembros públicos.*



### Observación de ingeniería de software 26.4

*Mantenga privadas todas las variables de instancia. Cuando sea necesario, proporcione métodos públicos para establecer los valores de variables de instancia privadas y para obtener los valores de variables de instancia privadas. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce los errores y mejora la posibilidad de modificación del programa.*



### Observación de ingeniería de software 26.5

*Los métodos tienden a caer en diversas categorías: métodos que obtienen los valores a partir de variables de instancia privadas; métodos que establecen los valores de las variables de instancia privadas; métodos que implementan los servicios de la clase; y métodos que realizan distintos mecanismos para la clase, tales como la inicialización de los objetos de las clases, la asignación de los objetos de las clases, y la conversión entre clases y los tipos predefinidos, o entre clases y otras clases.*

Los métodos de acceso pueden leer o desplegar datos. Otro uso común para los métodos de acceso es probar la verdad o falsedad de condiciones, por lo general, a dichos métodos se les llama *métodos predicados*. Un ejemplo de un método predicado podría ser el método **`estaVacia`** para cualquier clase contenedora; una clase capaz de almacenar muchos objetos, tales como una lista ligada, una pila o una cola. Un programa podría probar **`estaVacia`** antes de intentar leer otro elemento del objeto contenedor. Un programa podría probar **`estaLlena`** antes de intentar insertar otro elemento en el objeto contenedor.

La clase **`Hora1`** contiene los siguiente métodos públicos, **`Hora1`** (línea 14), **`estableceHora`** (línea 21), **`aCadenaUniversal`** (línea 29) y **`toString`** (línea 39). Éstos son *métodos públicos*, *servicios públicos* o la *interfaz de la clase*. Estos métodos los utilizan los *clientes* (es decir, porciones de un programa que son usuarios de una clase) de las clases para manipular los datos almacenados en los objetos de la clase.

Los clientes de una clase utilizan referencias para interactuar con objetos de la clase. Por ejemplo, el método **`paint`** dentro de un applet es un cliente de la clase **`Graphics`**; **`paint`** utiliza una referencia al objeto **`Graphics`** (tal como **`g`**), la cual lo recibe como argumento para dibujar en el applet, por medio de la llamada a los métodos que son servicios públicos de la clase **`Graphics`** (tales como **`drawString`**, **`drawLine`**, **`drawOval`** y **`drawRect`**).

Observe el método con el mismo nombre que la clase (línea 14); se trata del método *constructor* de la clase. Un constructor es un método especial que inicializa las variables de instancia del objeto de la clase. Se llama a un método constructor de la clase, cada vez que se crea la instancia de un objeto de dicha clase. Este constructor simplemente llama al método **estableceHora** de la clase (la cual explicaremos pronto) con los valores de la hora, el minuto y el segundo especificados como 0.

Los constructores pueden tomar argumentos pero no pueden devolver valor alguno. Una diferencia importante entre los constructores y otros métodos es que a los constructores *no se les permite especificar un tipo de dato de retorno* (incluso **void**). Por lo general, los constructores son métodos públicos de una clase. Más adelante explicaremos los métodos no públicos.



**Error común de programación 26.3**

*Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor, es un error lógico. Java permite a otros métodos de la clase tener el mismo nombre de la clase y especificar los tipos de retorno. Dichos métodos no son constructores y no se les llamará cuando se genere la instancia de un objeto de la clase.*

El método **estableceHora** (línea 21) es un método público que recibe tres argumentos enteros y los utiliza para establecer la hora. Cada argumento se prueba dentro de una expresión condicional que determina si el valor se encuentra en rango. Por ejemplo, el valor **hora** debe ser mayor que o igual que 0 y menor que 24, debido a que representamos el tiempo con formato de tiempo universal (0-23 para la hora, 0-59 para el minuto y 0-59 para el segundo). Cualquier valor fuera de este rango es un valor inválido y se establece en cero, lo que asegura que el objeto **Hora1** siempre contiene un dato válido. A esto se le llama *mantener al objeto en estado consistente*. En casos en los que se proporcionan datos inválidos para **estableceHora**, el programa podría querer indicar que se intentó establecer un valor inválido. Exploraremos esta posibilidad en los ejercicios.

**Buena práctica de programación 26.4**

*Siempre defina una clase de manera que sus variables de instancia se mantengan en un estado consistente.*

El método **aCadenaUniversal** (línea 29) no toma argumentos y devuelve un **String**. Este método produce una cadena con la hora en formato universal que consta de seis dígitos, dos para la hora, dos para el minuto y dos para el segundo. Por ejemplo, 13:30:07 representa 1:30:07 PM. La línea 31 crea una instancia de la clase **DecimalFormat** (del paquete **java.text** importado en la línea 3) para ayudar a mantener la hora en formato universal. El objeto **dosDigitos** se inicializa con la *cadena de control de formato* **"00"**, la cual indica que el formato del número debe consistir en dos dígitos, cada **0** es una posición para un dígito. Si el número al que se le da formato es de un solo dígito, a éste le antecede un **0** (es decir, a **8** se le da formato como **08**). La instrucción **return** de las líneas 33 a 35 utilizan el método **format** (que devuelve un **String** con formato, el cual contiene el número) del objeto **dosDigitos** para dar formato a los valores de la hora, el minuto y el segundo como cadenas de dos dígitos. Dichas cadenas se concatenan con el operador **+** (separado por punto y coma), y devuelto desde el método **aCadenaUniversal**.

El método **toString** (línea 29) no toma argumentos y devuelve un **String**. Este método produce una cadena con formato de hora estándar que consta de los valores **hora**, **minuto** y **segundo** separados por dos puntos y un indicador AM o PM, como en **1:27:06 PM**. Este método utiliza las mismas técnicas de **DecimalFormat** que el método **aCadenaUniversal**, para garantizar que los valores para **minuto** y **segundo** aparezcan con dos dígitos. El método **toString** es especial, debido a que heredamos un método **toString** de la clase **Object** con exactamente la primera línea que nuestro **toString** de la línea 39. El método **toString** original de la clase **Object** es una versión general que utilizamos con frecuencia como un contenedor que puede redefinirse mediante una subclase (similar a los métodos **init**, **start** y **paint** de la clase **JApplet**). Nuestra versión reemplaza a la versión que heredemos para proporcionar un método **toString** más apropiado para nuestra clase. A esto se le conoce como *redefinir* la definición original del método (explicada con detalle en el capítulo 27).

Una vez que se define la clase, ésta puede utilizarse como un tipo en una declaración como

```
Hora1 atardecer,       // referencia al objeto de tipo Hora1
      arregloHora[];   // referencia al arreglo de objetos de Hora1
```

El nombre de la clase es un nuevo especificador de tipo. Existen muchos objetos de una clase, así como pueden existir muchas variables de tipos de datos primitivos tales como **int**. El programador puede crear tantos

nuevos tipos de clases como sea necesario; ésta es una de las razones por las cuales a Java se le conoce como un *lenguaje extensible*.

La aplicación de la figura 26.1 (líneas 48 a 82) utiliza la clase **Hora1**. El método **main** de la clase **PruebaHora** declara e inicializa una instancia de la clase **Hora1** llamada **h** con la línea 56

```
Hora1 h = new Hora1();   // llama al constructor Hora1
```

Cuando se crea la instancia del objeto, el operador **new** asigna la memoria en la que se almacenará el objeto de **Hora1**, después **new** llama al constructor **Hora1** para inicializar las variables de instancia del nuevo objeto de **Hora1**. El constructor invoca al método **estableceHora** para inicializar explícitamente cada variable de instancia privada en **0**. El operador **new** devuelve entonces una referencia al nuevo objeto, y dicha referencia se asigna a **h**. De manera similar, la línea 31 de la clase **Hora1** utiliza **new** para asignar la memoria para el objeto **DecimalFormat**, y luego llama al constructor **DecimalFormat** con el argumento "**00**" para indicar la cadena de control de formato del número.



### Observación de ingeniería de software 26.6

*Cada vez que **new** crea un objeto de la clase, se llama al constructor de dicha clase para inicializar las variables de instancia del nuevo objeto.*

Observe que la clase **Hora1** no se importó hacia  archivo **PruebaHora.java**. En realidad, cada clase en java es parte de un *paquete* (como las clases del API de JAVA). Si el programador no especifica el paquete para una clase, la clase se coloca automáticamente en el *paquete predeterminado*, el cual incluye las clases compiladas en el directorio actual. Si una clase se encuentra en el mismo paquete que el de la clase que la utiliza, no se requiere una instrucción **import**. Importamos clases desde el API de Java debido a que sus archivos **.class** no se encuentran en el mismo paquete con cada programa que escribimos. En la sección 26.4 explicamos cómo definir sus propios paquetes de clases.

La línea 57 declara una referencia a un **String** llamada **salida** que almacenará la cadena que contiene los resultados a desplegarse en el diálogo de mensaje. Las líneas 59 a 63 agregan la hora en formato universal a **salida** (al enviar un mensaje **aCadenaUniversal**, hacia el objeto al que hace referencia h) y en formato de tiempo estándar (al enviar el mensaje **toString**, hacia el objeto al que hace referencia h) para confirmar que los datos se inicializaron apropiadamente. Observe la línea 63

```
"\nLlamada implicita a toString(): " + h;
```

En Java, el operador **+** puede utilizarse para concatenar cadenas. Aplicar el operador **+** a una cadena y a un objeto da como resultado una llamada implícita al método **toString** del objeto, el cual convierte al objeto en una cadena. El operador **+** después concatena las dos cadenas para producir una sola. Las líneas 62 y 63 muestran que usted puede llamar a **toString** tanto implícita como explícitamente, en una operación de concatenación de cadenas.

La línea 65

```
h.estableceHora( 13, 27, 6 );
```

envía el mensaje **estableceHora** al objeto al cual **h** hace referencia, para modificar nuevamente la hora de **salida** en ambos formatos y confirmar que la hora se estableció correctamente.

Para mostrar que el método **estableceHora** valida los valores que se le pasan, la línea 71

```
h.estableceHora( 99, 99, 99 );   // todos son valores inválidos
```

llama al método **estableceHora** e intenta establecer las variables de instancia con valores válidos. Luego, las líneas 72 a 74 agregan nuevamente la hora a **salida** en ambos formatos para confirmar que **estableceHora** valida los datos. Las líneas 76 a 78 despliegan un cuadro de mensaje con los resultados de nuestro programa. Observe en las dos últimas líneas de la ventana de salida que la hora se establece en medianoche; el valor predeterminado del objeto **Hora1**.

Ahora que hemos visto nuestra primera clase que no es un applet ni una aplicación, consideremos varios puntos del diseño de clases.

De nuevo, observe que las variables de instancia **hora**, **minuto** y **segundo** se declaran en donde se definen. Aquí, la filosofía es que la representación de los datos reales utilizada dentro de las clases no es asunto de los clientes de la clase. Por ejemplo, sería perfectamente razonable para la clase representar la hora interna-

mente como el número de segundos desde medianoche. Los clientes podrían utilizar los mismos métodos públicos y obtener los mismos resultados sin darse cuenta de esto. En este sentido, se dice que la implementación de una clase está *oculta* a sus clientes. El ejercicio 26.10 le pide que haga las modificaciones precisas a la clase **`Hora1`** de la figura 26.1 para mostrar que no existe un cambio visible para los clientes de la clase.



### Observación de ingeniería de software 26.7

*El ocultamiento de información promueve la capacidad de modificación del programa y simplifica la percepción de los clientes respecto a la clase.*



### Observación de ingeniería de software 26.8

*Los clientes de una clase pueden (y deben) utilizar la clase sin conocer los detalles de implementación de la clase. Si cambia la implementación de la clase (por ejemplo, para mejorar el rendimiento), la interfaz proporcionada permanece constante, el código fuente de los clientes de la clase no necesitan modificación. Esto hace mucho más fácil la modificación de los sistemas.*

En este programa, el constructor **`Hora1`** simplemente inicializa las variables de instancia en 0 (es decir, el equivalente militar de las 12 AM). Esto asegura que el objeto se crea con un *estado consistente* (es decir, todos los valores de las variables de instancia son válidos). Los valores no válidos no pueden almacenarse en las variables de instancia del objeto **`Hora1`** debido a que el constructor se llama cuando se crea el objeto **`Hora1`**, y los intentos subsiguientes de un cliente por modificar las variables de instancia se examinan mediante el método **`estebleceHora`**.

Las variables de instancia pueden inicializarse cuando se declaran en el cuerpo de la clase, por medio del constructor de la case, o se les puede asignar valores por medio de instrucciones *establecer*. Las variables de instancia que el programador no inicializa explícitamente, las inicializa el compilador (las variables numéricas primitivas se establecen en 0, las booleanas en **`false`** y las referencias se establecen en **`null`**).



### Buena práctica de programación 26.5

*Inicialice las variables de instancia de una clase en el constructor de esa clase.*

Es interesante que los métodos **`aCadenaUniversal`** y **`toString`** no tomen argumentos. Esto se debe a que estos métodos saben implícitamente que van a manipular las variables de instancia del objeto **`Hora1`** particular para el que se invocaron. Esto hace las llamadas a los métodos más concisas que las llamadas convencionales a funciones en la programación por procedimientos. También reduce la probabilidad de pasar los argumentos incorrectos, los tipos incorrectos de los argumentos y/o el número incorrecto de argumentos, como sucede con frecuencia en las llamadas a funciones en C.



### Observación de ingeniería de software 26.9

*Con frecuencia, utilizar un método de programación orientada a objetos simplifica las llamadas a los métodos, al reducir el número de parámetros a pasar. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de las variables de instancia y de los métodos dentro de un objeto le da a los métodos el derecho de acceso a las variables de instancia.*

Las clases simplifican la programación debido a que el cliente (o usuario del objeto de la clase) solamente necesitan preocuparse por las operaciones públicas encapsuladas en el objeto. Por lo general, dichas operaciones están diseñadas para que estén orientadas al cliente en lugar de a la implementación. Los clientes no necesitan preocuparse por la implementación de la clase. La interfaz cambia, pero con menos frecuencia que las implementaciones. Cuando la implementación cambia, el código que depende de la implementación debe cambiar en concordancia. Al ocultar la implementación eliminamos la posibilidad de que otras partes del programa se hagan dependientes de los detalles de la implementación de la clase.

Con frecuencia, las clases no tienen que crearse "desde cero". En lugar de eso, pueden *derivarse* desde otras clases que proporcionan operaciones que las nuevas clases pueden utilizar, o las clases pueden incluir como miembros objetos de otras clases. Tal *reutilización de software* puede mejorar enormemente la productividad del programador. A la derivación de clases a partir de clases existentes se le llama *herencia* y la explicaremos con detalle en el capítulo 27. A la inclusión de objetos de clases como miembros de otras clases se le llama *composición* o *agregación*, y la explicaremos más adelante en este capítulo.

## 26.3 Alcance de una clase

Las variables de instancia y los métodos de una clase pertenecen al *alcance de dicha clase*. Dentro del alcance de dicha clase, los miembros están accesibles de inmediato para todos los métodos de la clase y se puede hacer referencia a ellos simplemente por su nombre. Fuera del alcance de la clase, no se puede hacer referencia a los miembros de la clase directamente por su nombre. Sólo se puede acceder a dichos miembros de la clase (tales como miembros públicos) que son visibles por medio de un "manipulador" (es decir, se puede hacer referencia a los miembros con un tipo de dato primitivo por medio de **nombreReferenciaObjeto.nombreVariablePrimitiva**, y se puede hacer referencia a los miembros del objeto por medio de **nombreReferenciaObjeto.nombreMiembroObjeto**).

Las variables definidas en un método sólo se conocen en dicho método (es decir, son variables locales a dicho método). Se dice que dichas variables tienen alcance de bloque. Si un método define una variable con el mismo nombre que la variable con alcance de clase (es decir, una variable de instancia), la variable con alcance de clase se oculta en la variable local con alcance de método. Se puede acceder a una variable de instancia oculta en un método, al anteceder a su nombre la palabra reservada **this** y el operador punto, como en **this.x**. Más adelante en este capítulo, explicaremos la palabra reservada **this**.

## 26.4 Creación de paquetes

Como hemos visto en casi cada ejemplo del libro, las clases y las *interfaces* (que explicaremos en el capítulo 27) de las bibliotecas existentes, tales como la API de Java, pueden importarse dentro de un programa en Java. Cada clase e interfaz del API de Java pertenece a un paquete específico que contiene un grupo de clases e interfaces relacionadas. En realidad, los paquetes son estructuras de directorios que se utilizan para organizar las clases y las interfaces. Los paquetes proporcionan un mecanismo para la *reutilización de software*. Una de las metas de los programadores es crear componentes reutilizables de software, de manera que no sea necesario redefinir el código repetidamente en cada programa. Otro beneficio de los paquetes es que proporcionan una convención para los *nombres de clase únicos*. Con cientos de miles de programas en Java alrededor del mundo, existen muchas posibilidades de que los nombres que usted elija para las clases tengan conflicto con los nombres que otros programadores utilizan para sus clases.

La aplicación de la figura 26.2 ilustra la manera de crear su propio paquete y cómo utilizar una clase a partir de dicho paquete dentro de un programa.

```
// Figura 26.2: Hora1.java
// Definición de la clase Hora1
package com.deitel.chtp4.Cap26;
import java.text.DecimalFormat;  // utilizado para dar formato al número

// Esta clase mantiene la hora en formato de 24 horas
public class Hora1 extends Object {
   private int hora;     // 0 - 23
   private int minuto;   // 0 - 59
   private int segundo;  // 0 - 59

   // El constructor Hora1 inicializa en cero cada variable
   // de instancia. Garantiza que cada objeto Hora1 inicia en un
   // estado consistente.
   public Hora1()
   {
      estableceHora( 0, 0, 0 );
   } // fin del constructor Hora1

   // Establece un nuevo valor de hora utilizando la hora militar. Realiza
```

**Figura 26.2** Creación de un paquete para reutilización de software; **Hora1.java**. (Parte 1 de 2.)

```
    // validaciones de datos. Establece en cero a los
    // valores inválidos.
    public void estableceHora( int h, int m, int s )
    {
       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
    } // fin del método estableceHora

    // Convierte a String en hora de formato universal
    public String aCadenaUniversal()
    {
       DecimalFormat dosDigitos = new DecimalFormat( "00" );

       return dosDigitos.format( hora ) + ":" +
              dosDigitos.format( minuto ) + ":" +
              dosDigitos.format( segundo );
    } // fin del método aCadenaUniversal

    // Convierte a String en hora de formato estándar
    public String toString()
    {
       DecimalFormat dosDigitos = new DecimalFormat( "00" );

       return ( (hora == 12 || hora == 0) ? 12 : hora % 12 ) +
              ":" + dosDigitos.format( minuto ) +
              ":" + dosDigitos.format( segundo ) +
              ( hora < 12 ? " AM" : " PM" );
    } // fin del método toString
} // fin de la clase Hora1
```

**Figura 26.2** Creación de un paquete para reutilización de software; **`Hora1.java`**. (Parte 2 de 2.)

```
// Figura 26.2: PruebaHora.java
// Clase PruebaHora para utilizar la clase importada Hora1
import javax.swing.JOptionPane;
import com.deitel.chtp4.Cap26.Hora1;  // importa a la clase Hora1

public class PruebaHora {
   public static void main( String args[] )
   {
      Hora1 h = new Hora1();

      h.estableceHora( 13, 27, 06 );
      String salida =
         "La hora universal es: " + h.aCadenaUniversal() +
         "\nLa hora estandar es: " + h.toString();

      JOptionPane.showMessageDialog( null, salida,
         "Empacando la clase Hora1 para reutilizarla",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
```

**Figura 26.2** Creación de un paquete para reutilización de software; **`PruebaHora.java`**. (Parte 1 de 2.)

```
   } // fin de main
} // fin de la clase PruebaHora
```

Empacando la clase Hora1 para reutilizarla

La hora universal es: 13:27:06

La hora estandar es: 1:27:06 PM

Aceptar

**Figura 26.2** Creación de un paquete para reutilización de software; **`PruebaHora.java`**. (Parte 2 de 2.)

Los pasos para crear clases reutilizables son:

1. Declare una clase pública. Si la clase no es pública, solamente puede ser utilizada por otras clases en el mismo paquete.
2. Elija un nombre de paquete, y agregue una *instrucción* ***`package`*** al archivo de código fuente para la declaración de la clase reutilizable. Solamente puede haber una instrucción **`package`** en el archivo de código fuente de Java, y debe anteceder a todas las demás declaraciones e instrucciones en el archivo.
3. Compile la clase de manera que se coloque en el lugar apropiado de la estructura de directorio del paquete.
4. Importe la clase reutilizable dentro de un programa, y utilice la clase.

Para el *paso 1*, elegimos utilizar la clase pública **`Hora1`** de la figura 26.1. No hicimos modificaciones a la implementación de la clase, de modo que no explicaremos nuevamente los detalles de implementación de dicha clase.

Para satisfacer el *paso 2*, agregamos una instrucción **`package`** al principio del archivo. La línea 3

```
package com.deitel.chtp4.Cap26;
```

utiliza la *instrucción* **`package`** para definir un paquete con el nombre **`com.deitel.chtp4.Cap26`**. Al colocar la instrucción **`package`** al principio del archivo de código fuente en Java indicamos que la clase definida en el archivo es parte del paquete especificado. Las únicas instrucciones en Java que aparecen fuera de las llaves de la definición de la clase son las instrucciones **`package`** e **`import`**.



**Observación de ingeniería de software 26.10**

*Un archivo de código fuente en Java tiene el siguiente orden: una instrucción* ***`package`*** *(si existe alguna), instrucción* ***`import`*** *(si existen), y las definiciones de las clases. Solamente una de las definiciones de las clases puede ser pública. Las demás clases en el archivo también se colocan en el paquete, pero no son reutilizables. Éstas se encuentran en el paquete para soportar a la clase reutilizable del archivo.*

En un esfuerzo por proporcionar un nombre único para cada paquete. Sun Microsystems especifica una convención para asignar nombres a los paquetes. Cada nombre de paquete debe comenzar con el nombre de su dominio de Internet en orden inverso. Por ejemplo, nuestro dominio de Internet es **`deitel.com`**, de modo que el nombre de nuestro paquete inicia como **`com.deitel`**. Si su nombre de dominio es *suescuela***`.edu`** el nombre del paquete que usted utilizaría es **`edu.`***suescuela*. Después de invertir el nombre de dominio, puede elegir cualquier nombre que desee para su paquete. Si usted forma parte de una empresa con muchas divisiones, o de una universidad con muchas escuelas, podría utilizar el nombre de su división o escuela como el siguiente nombre del paquete. Elegimos utilizar **`chtp4`** como el siguiente nombre de nuestro paquete para indicar que esta clase es parte del libro. El último nombre en nuestro paquete especifica que es para el capítulo 26 (**`Cap26`**). [*Nota:* Utilizaremos nuestros propios paquetes a lo largo del libro. Usted puede determinar el capítulo en el que nuestras clases reutilizables están definidas, observando el último nombre de la instrucción **`import`**.]

El *paso 3* consiste en compilar la clase para almacenarla en el paquete apropiado. Cuando se compila un archivo en Java que contiene una instrucción **`package`**, el archivo de clase que resulta se coloca en la estructura de directorio especificada por la instrucción **`package`**. La instrucción **`package`** de la figura 26.2 indica que la clase **`Hora1`** debe colocarse en el directorio **`Cap26`**. Los otros nombres, **`com`**, **`deitel`** y **`chtp4`**, tam-

bién son directorios. Los nombres de directorios en la instrucción **package** especifican la ubicación exacta de las clases en el paquete. Si estos directorios no existen antes de la compilación de la clase, el compilador los crea.

Cuando se compila una clase dentro de un paquete, la opción **(*-d*)** de la línea de comando provoca que el compilador **javac** genere los directorios apropiados, basándose en la instrucción **package** de la clase. Además, la opción especifica en dónde crear (o localizar) los directorios. Por ejemplo, en una ventana de comando, utilizamos el comando de compilación

```
javac –d . Hora1.java
```

para especificar que el primer directorio de nuestro paquete debe colocarse en el directorio actual. El **.** después de **-d** del comando anterior representa el directorio actual en los sistemas operativos Windows, UNIX y Linux (y muchos otros también). Después de ejecutar el comando de compilación, el directorio actual contiene un directorio llamado **com**; **com** contiene un directorio llamado **deitel**; **deitel** contiene un directorio llamado **chtp4**, y **chtp4** contiene un directorio llamado **Cap26**. En el directorio **Cap26** puede encontrar el archivo **Hora1.class**. [*Nota:* Si no utiliza la opción **-d**, entonces primero debe copiar o mover el archivo de la clase al directorio de paquete apropiado después de compilarlo.]

El nombre del paquete es parte del nombre de la clase. El nombre de la clase en este ejemplo es en realidad **com.deitel.chtp4.Cap26.Hora1**. Usted puede utilizar este nombre *completo* en sus programas, o puede importar la clase y utilizarla con su nombre simple (**Hora1**) en el programa. Si otro paquete también contiene una clase **Hora1**, se puede utilizar el nombre completo de la clase para distinguir entre las clases y evitar un *conflicto de nombres* (también llamado *colisión de nombres*).

Una vez que la clase se compila y se almacena en el paquete, ésta puede importarse dentro de los programas (*Paso 4*). La línea 54

```
import com.deitel.chtp4.Cap26.Hora1;   // importa la clase Hora1
```

especifica que la clase **Hora1** debe importarse para utilizarla en la clase **PruebaHora**. [*Nota:* Las clases del paquete nunca necesitan importar otras clases del mismo paquete.]

## 26.5 Inicialización de los objetos de una clase: Constructores

Cuando se crea un objeto, sus miembros pueden inicializarse por medio de un método *constructor*. Un constructor es un método con el mismo nombre que la clase (con sensibilidad a mayúsculas y minúsculas). El programador proporciona el constructor que se invoca de manera automática cada vez que se crea la instancia de un objeto de la clase. Las variables de instancia pueden inicializarse implícitamente con sus valores predeterminados (**0** para los tipos numéricos primitivos, **false** para los booleanos y **null** para las referencias), y pueden inicializarse en el constructor de la clase, o posteriormente a la creación del objeto. Los constructores no pueden especificar tipos de retorno o valores de retorno. Una clase puede contener constructores sobrecargados para proporcionar los medios para inicializar los objetos de dicha clase.



**Buena práctica de programación 26.6**

*Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicializa apropiadamente con valores significativos.*

Cuando se crea un objeto de una clase, los *inicializadores* pueden proporcionarse entre paréntesis a la derecha del nombre de la clase. Estos inicializadores se pasan como argumentos al constructor de la clase. En el siguiente ejemplo demostraremos esta técnica. También hemos visto esta técnica varias veces antes, cuando creamos nuevos objetos de clases como **DecimalFormat**, **JLabel**, **JTextField**, **JTextArea** y **JButton**. Para cada una de estas clases vimos instrucciones de la forma

```
ref = new NombreClase( argumentos );
```

en donde **ref** es una referencia al tipo de dato apropiado; **new** indica la creación del nuevo objeto; *NombreClase* indica el tipo del nuevo objeto, y *argumentos* especifica los valores utilizados por el constructor de la clase para inicializar al objeto.

Si no se definen constructores para la clase, el compilador crea un *constructor predeterminado* que no toma argumentos (también llamado *constructor sin argumentos*).El *constructor predeterminado* de una clase

llama al constructor predeterminado de la clase a la cual extiende, luego procede a inicializar las variables de instancia de la manera en que explicamos anteriormente (es decir, las variables numéricas primitivas en 0, las booleanas en **`false`** y las referencias en **`null`**). Si la clase que extiende a esta clase no contiene un constructor predeterminado, el compilador emite un mensaje de error. También es posible que los programadores proporcionen un constructor sin argumentos como lo mostramos con la clase **`Hora1`** y que veremos en el siguiente ejemplo. Si el programador define un constructor, Java no creará el constructor predeterminado para la clase.

**Error común de programación 26.4**

*Si se proporcionan los constructores para la clase, pero ninguno de los constructores públicos es un constructor sin argumentos, y se intenta hacer una llamada al constructor sin argumentos para inicializar un objeto de la clase, ocurre un error de sintaxis. Es posible llamar a un constructor sin argumentos solamente si no existen constructores para esa clase (se llama al constructor predeterminado), o si no existe un constructor sin argumentos.*

## 26.6 Uso de los métodos *obtener* y *establecer*

Las variables de instancia privadas pueden manipularse únicamente a través de los métodos de la clase. Una manipulación común podría ser el ajuste del saldo de un cliente en el banco (por ejemplo, una variable de instancia de la clase **`CuentaBanco`**) por medio de un método **`calculaInteres`**.

Con frecuencia, las clases proporcionan métodos públicos para permitir a los clientes de la clase *establecer* u *obtener* variables de instancia privadas. Estos métodos no necesitan llamarse *establecer* u *obtener*, pero con frecuencia se llaman así. Si usted realiza un estudio más profundo de Java verá que la convención de nombres es importante para crear componentes de software reutilizable en llamados *JavaBeans*.

Como un ejemplo de nomenclatura, un método que establece la variable de instancia **`tasaInteres`** por lo general se escribiría como **`estableceTasaInteres`** y el método que obtiene **`tasaInteres`** por lo general se llamaría **`obtieneTasaInteres`**. Por lo general, a los métodos *obtener* también se les conoce como *métodos de acceso* o *métodos de consulta*. Por lo general, a los métodos *establecer* también se les conoce como *métodos de mutación* (debido a que por lo general modifican un valor).

Podría parecer que proporcionar las capacidades de las funciones *obtener* y *establecer* es, en esencia, lo mismo que hacer públicas las variables de instancia. Ésta es otra sutileza de Java que hace al lenguaje tan apropiado para la ingeniería de software. Si una variable de instancia es pública, puede leerse o escribirse en dicha variable de instancia por medio de cualquier método del programa. Si una variable de instancia es privada, ciertamente parecería que un método *obtener* permitiría a otros métodos leer sus datos, pero el método *obtener* controla el formato y el desplegado de los datos. Un método *establecer* público puede, y muy probablemente lo hará, intentar hacer un cuidadoso escrutinio para modificar el valor de la variable de instancia. Esto garantiza que el nuevo valor es apropiado para dicho elemento de dato. Por ejemplo, intentar *establecer* un día del mes para una fecha con día 37 será rechazado, intentar *establecer* el peso de una persona en un valor negativo será rechazado, y así sucesivamente. Por lo tanto, aunque los métodos *establecer* y *obtener* pueden proporcionar acceso a datos privados, el programador restringe el acceso por medio de la implementación de los métodos.

Los beneficios de la integridad de datos no son automáticos sencillamente porque las variables de instancia se hagan privadas; el programador debe proporcionar las validaciones necesarias. Java proporciona el marco de trabajo en el que los programadores pueden diseñar mejores programas de manera más conveniente.



**Observación de ingeniería de software 26.11**

*Los métodos que establecen los valores de datos privados deben verificar que los nuevos valores que se pretenden sean apropiados; si no lo son, los métodos establecer deben colocar las variables de instancia privadas en un estado consistente apropiado.*

Los métodos *establecer* de una clase pueden devolver valores que indiquen que se hicieron intentos para asignar datos no válidos a los objetos de la clase. Esto permite a los clientes de la clase verificar los valores de retorno de los métodos *establecer* para determinar si los objetos que manipulan son válidos, y tomar la decisión adecuada si no lo son.



**Buena práctica de programación 26.7**

*Todo método que modifica las variables de instancia privadas de un objeto deben asegurarse de que los datos permanecen en un estado consistente.*

El applet de la figura 26.3 mejora nuestra clase **Hora** (ahora llamada **Hora2**) para que incluya los métodos *obtener* y *establecer* para las variables de instancia privadas **hora**, **minuto** y **segundo**. Los métodos *establecer* controlan de manera estricta el establecimiento de las variables de instancia en valores válidos. Intentar establecer una variable de instancia en un valor incorrecto provoca que la variable de instancia se establezca en cero (y la deja en un estado consistente). Cada método *obtener* simplemente devuelve el valor apropiado de las variables de instancia. Este applet además introduce las técnicas avanzadas de manipulación de eventos GUI al comenzar con la definición de nuestra primera aplicación completa con ventanas.

```
// Figura 26.3: Hora2.java
// Definición de la clase Hora2
package com.deitel.chtp4.Cap26;   // coloca a Hora2 en un paquete
import java.text.DecimalFormat;   // utilizado para dar formato al número

// Esta clase mantiene la hora en formato de 24 horas
public class Hora2 extends Object {
   private int hora;      // 0 - 23
   private int minuto;    // 0 - 59
   private int segundo;   // 0 - 59

   // El constructor Hora2 inicializa en cero a cada
   // variable de instancia. Garantiza que el objeto Hora inicia en un
   // estado consistente.
   public Hora2() { estableceHora( 0, 0, 0 ); }

   // Métodos establecer
   // Establece un nuevo valor de hora por medio del horario universal.
   // Realiza validaciones de datos. Establece en cero a los valores
      inválidos.
   public void estableceHora( int h, int m, int s )
   {
      estableceHora( h );     // establece la hora
      estableceMinuto( m );   // establece el minuto
      estableceSegundo( s );  // establece el segundo
   } // fin del método estableceHora

   // establece la hora
   public void estableceHora( int h )
      { hora = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

   // establece el minuto
   public void estableceMinuto( int m )
      { minuto = ( ( m >= 0 && m < 60 ) ? m : 0 ); }

   // establece el segundo
   public void estableceSegundo( int s )
      { segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

   // Métodos obtener
   // obtiene la hora
   public int obtieneHora() { return hora; }

   // obtiene el minuto
   public int obtieneMinuto() { return minuto; }

   // obtiene el segundo
```

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **Hora2.java**. (Parte 1 de 2.)

```
   public int obtieneSegundo() { return segundo; }

   // Convierte a String en hora en formato universal
   public String aCadenaUniversal()
   {
      DecimalFormat dosDigitos = new DecimalFormat( “00” );

      return dosDigitos.format( obtieneHora() ) + “:” +
             dosDigitos.format( obtieneMinuto() ) + “:” +
             dosDigitos.format( obtieneSegundo() );
   } // fin del método aCadenaUniversal

   // Convierte a String en hora en formato estándar
   public String toString()
   {
      DecimalFormat dosDigitos = new DecimalFormat( “00” );

      return ( ( obtieneHora() == 12 || obtieneHora() == 0 ) ?
                   12 : obtieneHora() % 12 ) + “:” +
             dosDigitos.format( obtieneMinuto() ) + “:” +
             dosDigitos.format( obtieneSegundo() ) +
              ( obtieneHora() < 12 ? “ AM” : “ PM” );
   } // fin del método toString
} // fin de la clase Hora2
```

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **`Hora2.java`**. (Parte 2 de 2.)

```
// Figura 26.3: PruebaHora.java
// Demostración de los métodos establecer y obtener de la clase Hora2
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.deitel.chtp4.Cap26.Hora2;

public class PruebaHora extends JApplet
                        implements ActionListener {
   private Hora2 h;
   private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
   private JTextField campoHora, campoMinuto,
                        campoSegundo, despliega;
   private JButton botonMarcar;

   public void init()
   {
      h = new Hora2();

      Container c = getContentPane();

      c.setLayout( new FlowLayout() );
      etiquetaHora = new JLabel( “Establece la hora” );
      campoHora = new JTextField( 10 );
      campoHora.addActionListener( this );
      c.add( etiquetaHora );
      c.add( campoHora );

```

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **`PruebaHora.java`**. (Parte 1 de 4.)

```
      etiquetaMinuto = new JLabel( "Establece los minutos" );
      campoMinuto = new JTextField( 10 );
      campoMinuto.addActionListener( this );
      c.add( etiquetaMinuto );
      c.add( campoMinuto );

      etiquetaSegundo = new JLabel( "Establece los segundos" );
      campoSegundo = new JTextField( 10 );
      campoSegundo.addActionListener( this );
      c.add( etiquetaSegundo );
      c.add( campoSegundo );

      despliega = new JTextField( 30 );
      despliega.setEditable( false );
      c.add( despliega );

      botonMarcar = new JButton( "Agrega 1 a segundo" );
      botonMarcar.addActionListener( this );
      c.add( botonMarcar );

      actualizadespliega();
   } // fin del método init

   public void actionPerformed( ActionEvent e )
   {
      if ( e.getSource() == botonMarcar )
         marca();
      else if ( e.getSource() == campoHora ) {
         h.estableceHora(
            Integer.parseInt( e.getActionCommand() ) );
         campoHora.setText( "" );
      }
      else if ( e.getSource() == campoMinuto ) {
         h.estableceMinuto(
            Integer.parseInt( e.getActionCommand() ) );
         campoMinuto.setText( "" );
      }
      else if ( e.getSource() == campoSegundo ) {
         h.estableceSegundo(
            Integer.parseInt( e.getActionCommand() ) );
         campoSegundo.setText( "" );
      }

      actualizadespliega();
   } // fin del método actionPerformed

   public void actualizadespliega()
   {
      despliega.setText( "Hora: " + h.obtieneHora() +
         "; Minuto: " + h.obtieneMinuto() +
         "; Segundo: " + h.obtieneSegundo() );
      showStatus( "La hora estandar es: " + h.toString() +
         "; La hora universal es: " + h.aCadenaUniversal() );
   } // fin del método actualizadespliega

   public void marca()
```

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **`PruebaHora.java`**. (Parte 2 de 4.)

```
   {
      h.estableceSegundo( ( h.obtieneSegundo() + 1 ) % 60 );

      if ( h.obtieneSegundo() == 0 ) {
         h.estableceMinuto( ( h.obtieneMinuto() + 1 ) % 60 );

         if ( h.obtieneMinuto() == 0 )
            h.estableceHora( ( h.obtieneHora() + 1 ) % 24 );
      } // fin de if
   } // fin del método marca
} // fin de la clase PruebaHora
```

AppletViewer: PruebaHora.class
Subprograma
Establece la hora 23 Establece los minutos
Establece los segundos
Hora: 0; Minuto: 0; Segundo: 0
Agrega 1 a segundo
Subprograma iniciado.

AppletViewer: PruebaHora.class
Subprograma
Establece la hora Establece los minutos
Establece los segundos
Hora: 23; Minuto: 0; Segundo: 0
Agrega 1 a segundo
La hora estandar es: 11:00:00 PM; La hora universal es: 23:00:00

AppletViewer: PruebaHora.class
Subprograma
Establece la hora Establece los minutos 59
Establece los segundos
Hora: 23; Minuto: 0; Segundo: 0
Agrega 1 a segundo
La hora estandar es: 11:00:00 PM; La hora universal es: 23:00:00

AppletViewer: PruebaHora.class
Subprograma
Establece la hora Establece los minutos
Establece los segundos
Hora: 23; Minuto: 59; Segundo: 0
Agrega 1 a segundo
La hora estandar es: 11:59:00 PM; La hora universal es: 23:59:00

AppletViewer: PruebaHora.class
Subprograma
Establece la hora Establece los minutos
Establece los segundos 58
Hora: 23; Minuto: 59; Segundo: 0
Agrega 1 a segundo
La hora estandar es: 11:59:00 PM; La hora universal es: 23:59:00

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **`PruebaHora.java`**. (Parte 3 de 4.)

AppletViewer: PruebaHora.class
Subprograma
Establece la hora | Establece los minutos
Establece los segundos
Hora: 23; Minuto: 59; Segundo: 58
Agrega 1 a segundo
La hora estandar es: 11:59:58 PM; La hora universal es: 23:59:58

AppletViewer: PruebaHora.class
Subprograma
Establece la hora | Establece los minutos
Establece los segundos
Hora: 23; Minuto: 59; Segundo: 59
Agrega 1 a segundo
La hora estandar es: 11:59:59 PM; La hora universal es: 23:59:59

AppletViewer: PruebaHora.class
Subprograma
Establece la hora | Establece los minutos
Establece los segundos
Hora: 0; Minuto: 0; Segundo: 0
Agrega 1 a segundo
La hora estandar es: 12:00:00 AM; La hora universal es: 00:00:00

**Figura 26.3** Uso de los métodos *establecer* y *obtener;* **`PruebaHora.java`**. (Parte 4 de 4.)

Los nuevos métodos *establecer* de la clase se definen en las líneas 28, 32 y 36 respectivamente. Observe que cada método realiza la misma instrucción condicional que estaba previamente en el método **`estableceHora`** para establecer la **`hora`**, el **`minuto`** y el **`segundo`**. Con la adición de estos métodos fuimos capaces de redefinir el cuerpo del método **`estableceHora`** (línea 20) para utilizar estos tres métodos y establecer la hora.



### Observación de ingeniería de software 26.12

*Si un método de la clase proporciona toda o parte de la funcionalidad requerida por otro método de la clase, llame a dicho método desde otro método. Esto simplifica el mantenimiento del código y reduce la probabilidad de error si la implementación del código se modifica. También es un ejemplo claro de la reutilización.*

Debido a las modificaciones en la clase **`Hora2`** que describimos antes, minimizamos las modificaciones que tienen que llevarse a cabo en la definición de la clase si la representación de los datos se modifica de **`hora`**, **`minuto`**, **`segundo`** a otra representación (tal como los segundos transcurridos durante el día). Solamente será necesario modificar los cuerpos de los métodos *establecer* y *obtener*. Esto permite al programador modificar la implementación de la clase sin afectar a los clientes de la misma clase (mientras los métodos públicos de la clase se llamen de la misma manera).

El applet **`PruebaHora`** proporciona una interfaz gráfica de usuario que permite al usuario ejecutar los métodos de la clase **`Hora2`**. El usuario puede establecer el valor de la hora, el minuto o el segundo al escribir un valor en el **`JTextField`** y oprimir la tecla *Entrar*. El usuario también puede hacer clic en el botón **Agrega 1 a segundo** para incrementar el tiempo en un segundo. En este applet, todos los eventos **`JTextField`** y **`JButton`** se procesan en el método **`actionPerformed`** (línea 122). Observe que las líneas 95, 101, 107 y 116 llaman a **`addActionListener`** para indicar que el applet debe comenzar a poner atención a **`campoHora`**, **`campoMinuto`**, **`campoSegundo`** de tipo **`JTextField`**, y a **`botonMarcar`** de tipo **`JButton`**, respectivamente. Además, observe que las cuatro llamadas utilizan **`this`** como argumento, lo que indica que el objeto de nuestra clase applet **`PruebaHora`** invoca a su **`actionPerformed`** para cada interacción con el usuario con estos compo-

nentes GUI. Esto provoca la siguiente interesante pregunta, ¿cómo determinamos el componente GUI con el que interactuó el usuario?

En **actionPerformed**, observe el uso de **e.getSource()** para determinar cuál componente GUI generó el evento. Por ejemplo, en la línea 124

```
if ( e.getSource() == botonMarcar )
```

determina si el usuario hizo clic en **botonMarcar**. Si es así, se ejecuta el cuerpo de la estructura **if**. De lo contrario, se evalúa la condición de la estructura **if** correspondiente a la línea 126, etcétera. Todo evento tiene una *fuente*, el componente GUI con el que el usuario interactuó para señalar al programa que realice una tarea. El parámetro **ActionEvent** que se le proporcionó a **actionPerformed** cada vez que ocurre el evento contiene una referencia hacia la fuente. La condición anterior simplemente pregunta, "¿la *fuente* del evento es **botonMarcar**?"

Después de cada operación, se despliega la hora resultante como una cadena, en la barra de estado del applet. Las ventanas de salida muestran al applet antes y después de las siguientes operaciones: establecer la hora en 23, establecer el minuto en 59, establecer el segundo en 58, e incrementar el segundo al doble con el botón **Agrega 1 a segundo**.

Observe que cuando se hace clic en el botón **Agrega 1 a segundo**, el método **actionPerformed** llama al método **marcar** (línea 154) del applet. El método **marcar** utiliza todos los nuevos métodos *obtener* y *establecer* para incrementar de manera apropiada los segundos. Aunque esto funciona, incurre en la sobrecarga de llamadas a múltiples métodos.



**Error común de programación 26.5**

*Un constructor puede llamar a otros métodos de la clase, tal como los métodos* establecer *y* obtener, *pero debido a que el constructor inicializa el objeto, las variables de instancia no pueden aún estar en un estado consistente. El uso de las variables de instancia antes de inicializarse de manera apropiada, es un error.*

Es verdad que los métodos *establecer* son importantes desde el punto de vista de la ingeniería de software, ya que pueden realizar validaciones. Los métodos *establecer* y *obtener* tienen otra ventaja en la ingeniería de software, como lo explicamos en la siguiente *Observación de ingeniería de software*.



**Observación de ingeniería de software 26.13**

*Acceder a los datos* **private** *a través de los métodos* establecer *y* obtener *no solamente protege a las variables de instancia de recibir valores no válidos, sino que además aísla a los clientes de la clase de la representación de las variables de instancia. Por lo tanto, si la representación de los datos cambia (por lo general, para reducir el almacenamiento requerido, o para mejorar el rendimiento), solamente necesita modificar las implementaciones del método; los clientes no necesitan modificación alguna mientras la interfaz proporcionada por los métodos permanezca igual.*

## 26.7 Uso de la referencia this

Cuando el método de una clase hace referencia a otro miembro de dicha clase para un objeto específico de la misma clase, ¿cómo asegura Java que se hace referencia al objeto apropiado? La respuesta es que cada objeto tiene acceso a una referencia a sí mismo, llamada *referencia* **this**.

La referencia **this** se utiliza implícitamente para hacer referencia tanto a las variables de instancia como a los métodos de un objeto. Por ahora, mostramos un ejemplo sencillo del uso explícito de la referencia **this**; más adelante, mostraremos algunos ejemplos sustanciales y sutiles del uso de **this**.



**Tip de rendimiento 26.2**

*Java conserva el almacenamiento, manteniendo sólo una copia de cada método por clase; este método es invocado por cada objeto de dicha clase. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase.*

La aplicación de la figura 26.4 muestra el uso implícito y explícito de la referencia **this** para permitir al método **main** de la clase **PruebaThis** desplegar los datos **private** del objeto **HoraSimple**.

La clase **HoraSimple** (líneas 20 a 46) define tres variables de instancia privadas, **hora**, **minuto** y **segundo**. El constructor (línea 23) recibe tres argumentos **int** para inicializar un objeto **HoraSimple**. Ob-

```
// Figura 26.4: PruebaThis.java
// Uso de la referencia this para hacer referencia a
// las variables de instancia y a los métodos.
import javax.swing.*;
import java.text.DecimalFormat;

public class PruebaThis {
   public static void main( String args[] )
   {
      HoraSimple h = new HoraSimple( 12, 30, 19 );

      JOptionPane.showMessageDialog( null, h.construyeCadena(),
         "Demostracion de la referencia \"this\" ",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin del método main
} // fin de la clase PruebaThis

class HoraSimple {
   private int hora, minuto, segundo;

   public HoraSimple( int hora, int minuto, int segundo )
   {
      this.hora = hora;
      this.minuto = minuto;
      this.segundo = segundo;
   } // fin del constructor HoraSimple

   public String construyeCadena()
   {
      return "this.toString(): " + this.toString() +
             "\ntoString(): " + toString() +
             "\nthis (con una llamada implicita a toString()): " +
             this;
   } // fin del método construyeCadena

   public String toString()
   {
      DecimalFormat dosDigitos = new DecimalFormat( "00" );

      return dosDigitos.format( this.hora ) + ":" +
             dosDigitos.format( this.minuto ) + ":" +
             dosDigitos.format( this.segundo );
   } // fin del método toString
} // fin de la clase HoraSimple
```

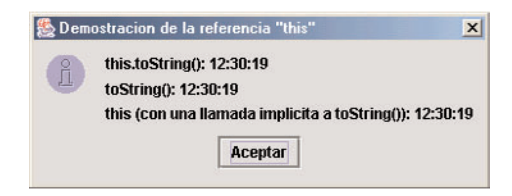



**Figura 26.4** Uso de la referencia `this`.

serve que los nombres de los parámetros para el constructor son los mismos que los nombres de las variables de instancia. Recuerde que una variable local de un método con el mismo nombre que una variable de instancia de la clase, oculta la variable de instancia en el alcance del método. Por esta razón, utilizamos la referencia **this** para hacer referencia explícita a la variable de instancia de las líneas 25 a 27.



**Error común de programación 26.6**

*En un método en el que un parámetro del método tiene el mismo nombre que uno de los miembros de la clase, utilice explícitamente* ***this*** *si quiere tener acceso al miembro de la clase; de lo contrario, hará una referencia incorrecta al parámetro del método.*



**Buena práctica de programación 26.8**

*Evite utilizar nombres de parámetros que tengan conflictos con los nombres de los métodos de las clases.*

El método **construyeCadena** (líneas 30 a 36) devuelve una **String** creada mediante la instrucción

```
return  “this.toString(): ” + this.toString() +
        “\ntoString(): ” + toString() +
        “\nthis (con una llamada implicita a toString()): ” +
        this;
```

la cual utiliza la referencia **this** de tres maneras. La primera línea invoca explícitamente el método **toString** de la clase, por medio de **this.toString()**. La segunda línea utiliza de manera implícita la referencia **this** para realizar la misma tarea. La tercera línea agrega **this** a la cadena que será devuelta. Recuerde que la referencia **this** es una referencia a un objeto; el objeto **HoraSimple** actual que se manipula. Como antes, cualquier referencia que se agrega a **String** da como resultado una llamada al método **toString** para el objeto referenciado. En la línea 12 se invoca el método **construyeCadena** para desplegar el resultado de las tres llamadas a **toString**. Observe que se despliega la misma hora en las tres líneas de salida, ya que las tres llamadas a **toString** son para el mismo objeto.

## 26.8 Finalizadores

Ya vimos que los métodos constructores son capaces de inicializar los datos de un objeto de la clase, cuando ésta se crea. Por lo general, los constructores adquieren recursos de sistema tales como memoria (cuando utiliza el comando **new**). Necesitamos una manera disciplinada de devolver los recursos al sistema cuando ya no son necesarios, para evitar el agotamiento de recursos. El recurso que más solicitan los constructores es la memoria. Java realiza la *recolección automática de basura* en la memoria para ayudar a devolver la memoria al sistema. Cuando un objeto ya no se utiliza en el programa (es decir, no existen referencias hacia el objeto), el objeto se *marca para el recolector de basura*. La memoria para dicho objeto puede reclamarse cuando se ejecuta el *recolector de basura*. Por tal motivo, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que la memoria no se reclama de manera automática en dichos lenguajes) son menos comunes en Java. Sin embargo, pueden ocurrir otras fugas de recursos.

Todas las clases en Java pueden tener un *método finalizador* que devuelva los recursos al sistema. Con seguridad, el método finalizador de un objeto será llamado para realizar la *limpieza final* en el objeto, justo antes de que el recolector de basura reclame la memoria del objeto. Un método finalizador de la clase siempre tiene el nombre **finalize**, no recibe parámetros y no devuelve valor alguno (es decir, su tipo de retorno es **void**). Una clase sólo puede tener un método **finalize** que no toma argumentos. El método **finalize** está definido originalmente en la clase **Object**, como un contenedor que no realiza acción alguna. Esto garantiza que cada clase tiene un método **finalize** para llamar al recolector de basura.

Hasta aquí, no hemos proporcionado finalizadores para las clases que hemos explicado. En realidad, los finalizadores rara vez se utilizan con clases sencillas. Veremos un ejemplo del método **finalize**, y explicaremos el recolector de basura más adelante en la figura 26.5.

## 26.9 Miembros estáticos de una clase

Cada objeto de una clase tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe compartirse una copia de una variable en particular entre todos los objetos de la clase. Una *variable*

*de clase* **static** se utiliza por ésta y por otras razones. Una variable de clase **static** representa información *para toda la clase*; todos los objetos de la clase comparten las mismas piezas de datos. La declaración de un miembro **static** comienza con la palabra reservada **static**.

Motivemos la necesidad de datos **static** para toda una clase con un ejemplo de juego de video. Suponga que tenemos un juego de video con **Marcianos** y otras criaturas del espacio. Cada **Marciano** tiende a ser más valiente y está más dispuesto a atacar a otras criaturas del espacio cuando el **Marciano** se da cuenta de que existen al menos cinco **Marcianos** presentes. Si existen menos de cinco **Marcianos** presentes, cada **Marciano** se acobarda. De modo que cada **Marciano** necesita conocer la **cuentaMarcianos**. Incluiremos a la clase **Marciano** el dato **cuentaMarcianos** como un dato de instancia. Si hacemos esto, entonces cada **Marciano** tendrá una copia separada del dato de instancia cada vez que creemos un nuevo **Marciano**, y no tendremos que actualizar la variable de instancia **cuentaMarcianos** en cada **Marciano**. Esto desperdicia espacio con copias redundantes y desperdicia tiempo en actualizar las copias separadas. En vez de lo anterior declaramos **cuentaMarcianos** como **static**. Esto hace a **cuentaMarcianos** un dato para toda la clase. Cada **Marciano** puede ver a **cuentaMarciano** como si fuera un dato de instancia del **Marciano**, pero solamente se mantiene una copia del **cuentaMarcianos** de tipo **static** en Java. Esto ahorra espacio. Ahorramos tiempo al incrementar **cuentaMarcianos static** del constructor de **Marciano**. Sólo existe una copia, de modo que no tenemos que incrementar copias separadas de **cuentaMarcianos** para cada objeto **Marciano**.



**Tip de rendimiento 26.3**

*Utilice las variables de clase* ***static*** *para ahorrar espacio, cuando sea suficiente una sola copia de los datos.*

Aunque las variables de clase **static** pueden parecer como variables globales, las variables de clase **static** tienen alcance de clase. Se puede acceder a los miembros de clase **public static** de la clase a través de una referencia a cualquier copia de la clase, o se puede acceder a ellas a través del nombre de la clase por medio del operador punto (por ejemplo, **Math.random()**). Se puede acceder a los miembros de clase **private static** de la clase a través de los métodos de la misma clase. En realidad, los miembros de clase **static** existen incluso cuando no existen objetos de dicha clase; están disponibles tan pronto como la clase se carga dentro de la memoria en tiempo de ejecución. Para acceder a un miembro de clase **private static** cuando no existen objetos de la clase, debe proporcionarse un método **public static** y el método debe invocarse colocando como prefijo el nombre de la clase y el operador punto.

El programa de la figura 26.5 muestra el uso de las variables de clase **private static** y de un método **public static**. La variable de clase **cuenta** se inicializa en cero de manera predeterminada. La variable de clase **cuenta** mantiene una cuenta del número de objetos de la clase **Empleado** que se instancia, y que actualmente reside en memoria. Esto incluye objetos que ya están señalados para el recolector de basura pero que aún no son reclamados.

```
// Figura 26.5: Empleado.java
// Declaración de la clase Empleado.
public class Empleado extends Object {
   private String nombre;
   private String apellido;
   private static int cuenta;  // # de objetos en memoria

   public Empleado( String nomb, String apell )
   {
      nombre = nomb;
      apellido = apell;

      ++cuenta;  // incrementa la cuenta estática de empleados
      System.out.println( “Constructor del objeto Empleado: “ +
```

**Figura 26.5** Uso de una variable de clase **static** para mantener la cuenta del número de objetos de una clase; **Empleado.java**. (Parte 1 de 2.)

```
                           nombre + “ “ + apellido );
   } // fin del constructor Empleado

   protected void finalize()
   {
      --cuenta;  // disminuye la cuenta estática de empleados
      System.out.println( “Finalizador del objeto Empleado: “ +
                           nombre + “ “ + apellido +
                           “; cuenta = “ + cuenta );
   } // fin del método finalize

   public String obtieneNombre() { return nombre; }

   public String obtieneApellido() { return apellido; }

   public static int obtieneCuenta() { return cuenta; }
} // fin de la clase Empleado
```

**Figura 26.5** Uso de una variable de clase **`static`** para mantener la cuenta del número de objetos de una clase; **`Empleado.java`**. (Parte 2 de 2.)

```
// Figura 26.5: PruebaEmpleado.java
// Prueba la clase empleado con una variable de clase estática,
// con un método de clase estática, y con memoria dinámica.
import javax.swing.*;

public class PruebaEmpleado {
   public static void main( String args[] )
   {
      String salida;

      salida = “Empleados antes de crear la instancia: “ +
                Empleado.obtieneCuenta();

      Empleado e1 = new Empleado( “Susana”, “Baez” );
      Empleado e2 = new Empleado( “Roberto”, “Jimenez” );

      salida += “\n\nEmpleados despues de crear la instancia: “ +
                 “\nvia e1.obtieneCuenta(): “ + e1.obtieneCuenta() +
                 “\nvia e2.obtieneCuenta(): “ + e2.obtieneCuenta() +
                 “\nvia Empleado.obtieneCuenta(): “ +
                 Empleado.obtieneCuenta();

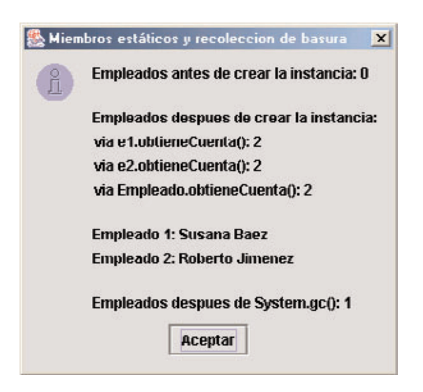
      salida += “\n\nEmpleado 1: “ + e1.obtieneNombre() +
                 “ “ + e1.obtieneApellido() +
                 “\nEmpleado 2: “ + e2.obtieneNombre() +
                 “ “ + e2.obtieneApellido();

      // marca los objetos a los que hace referencia e1 y e2
      // para recolección de basura
      e1 = null;
      e2 = null;

      System.gc(); // sugiere llamar al recolector de basura
```

**Figura 26.5** Uso de una variable de clase **`static`** para mantener la cuenta del número de objetos de una clase; **`PruebaEmpleado.java`**. (Parte 1 de 2.)

```
65
66          salida += "\n\nEmpleados despues de System.gc(): " +
67                     Empleado.obtieneCuenta();
68
69          JOptionPane.showMessageDialog( null, salida,
70             "Miembros estáticos y recoleccion de basura",
71             JOptionPane.INFORMATION_MESSAGE );
72          System.exit( 0 );
73       } // fin de main
74    } // fin de la clase PruebaEmpleado
```




```
Constructor del objeto Empleado: Susana Baez
Constructor del objeto Empleado: Roberto Jimenez
Finalizador del objeto Empleado: Susana Baez; cuenta = 1
Finalizador del objeto Empleado: Roberto Jimenez; cuenta = 0
```

**Figura 26.5** Uso de una variable de clase **`static`** para mantener la cuenta del número de objetos de una clase; **`PruebaEmpleado.java`**. (Parte 2 de 2.)

Cuando existen objetos de la clase **`Empleado`**, el miembro **`cuenta`** puede utilizarse en cualquier método de un objeto **`Empleado`**; en este ejemplo, el constructor incrementa la **`cuenta`** (línea 13) y el finalizador la disminuye (línea 20). Cuando no existen objetos de la clase **`Empleado`**, todavía se puede hacer referencia al miembro **`cuenta`**, pero solamente a través de una llamada al método **`public static obtieneCuenta`** de la siguiente manera:

```
Empleado.obtieneCuenta()
```

En este ejemplo, el método **`obtieneCuenta`** determina el número de objetos **`Empleado`** actualmente en memoria. Observe que cuando no existen objetos instanciados en el programa, se emite la llamada al método **`Empleado.obtieneCuenta()`**. Sin embargo, cuando existen instancias de objetos, el método **`obtieneCuenta`** también puede invocarse a través de una referencia a uno de los objetos, como en

```
e1.obtieneCuenta()
```



### Buena práctica de programación 26.9

*Siempre invoque métodos **`static`** por medio del nombre de la clase y del operador punto (**`.`**). Esto enfatiza a otros programadores que leen su código que el método que llaman es un método **`static`**.*

Observe que la clase **`Empleado`** tiene un método **`finalize`** (línea 18). Este método se incluye para mostrar cuándo se llama al recolector de basura en un programa. Por lo general, el método **`finalize`** se declara como ***`protected`***, de modo que no es parte de los servicios **`public`** de la clase. Explicaremos el modificador de acceso **`protected`** con detalle en el capítulo 27.

El método **`main`** de la aplicación **`PruebaEmpleado`** crea dos instancias del objeto **`Empleado`** (líneas 45 y 46). Cuando se invocan cada uno de los constructores del objeto **`Empleado`**, líneas 10 y 11, almacenan

referencias a los objetos **String** para el nombre y el apellido de dicho **Empleado**. Observe que estas dos instrucciones *no* hacen copias de los argumentos **String** originales. En realidad, los objetos **String** en Java son *inmutables*, éstos no pueden modificarse una vez creados (la clase **String** no proporciona método *establecer* alguno). Una referencia no puede utilizarse para modificar una **String**, de modo que es seguro hacer muchas referencias al objeto **String** en el programa en Java. Por lo general, éste no es el caso de la mayoría de las clases en Java.

Cuando **main** termina con los dos objetos **Empleado**, las referencias **e1** y **e2** se establecen en **null**, en las líneas 61 y 62. En este punto, las referencias **e1** y **e2** ya no hacen referencia a los objetos instanciados en las líneas 45 y 46. Esto marca a los objetos para el *recolector de basura*, debido a que no existen referencias a los objetos en el programa.

En algún momento, el recolector de basura reclama la memoria para estos casos (o el sistema operativo reclama la memoria cuando termina el programa). No existe certeza de cuándo actuará el recolector de basura, de modo que hacemos una llamada explícita al recolector de basura con la línea 64

```
System.gc;  // sugiere llamar al recolector de basura
```

la cual utiliza el método **public static gc** de la clase **System** (paquete **java.lang**), para sugerir la ejecución inmediata del recolector de basura. Sin embargo, ésta solamente es una sugerencia para la Java Virtual Machine (el intérprete); la sugerencia puede ignorarse. En nuestro ejemplo, el recolector de basura se ejecutó antes de que las líneas 69 a 71 desplegaran los resultados del programa. La última línea de la salida indica que el número de objetos **Empleado** en memoria es 1 después de llamar a **System.gc()**. Además, las dos últimas líneas de la salida en la ventana de comandos muestran que el objeto **Empleado** para **Susana Baez** se finalizó antes del objeto **Empleado** para **Roberto Jiménez**. El recolector de basura no garantiza la ejecución cuando se invoca a **System.gc()**, y no existe la garantía de que el recolector de basura recoja los objetos en un orden específico, de modo que es posible que la salida para este programa en su sistema puede diferir.

[*Nota:* Un método que se declara como **static** no puede acceder a miembros no estáticos de la clase. A diferencia de los métodos no estáticos, un método **static** no tiene referencia **this** debido a que las variables de clase estáticas y los métodos de clase estática existen independientemente de cualquier objeto de la clase y antes de que se genere cualquier instancia de un objeto de la clase.]



**Error común de programación 26.7**

*Hacer referencia a **this** en un método **static**, es un error de sintaxis.*





**Error común de programación 26.8**

*Es un error de sintaxis que un método **static** llame a un método de instancia o que acceda a una variable de instancia.*



**Observación de ingeniería de software 26.14**

*Cualquier variable de una clase **static** y cualquier método de una clase **static** puede utilizarse incluso si ningún objeto de esa clase se ha instanciado.*

## RESUMEN

- La POO encapsula los datos (atributos) y los métodos (comportamientos) dentro de objetos; los datos y los métodos de un objeto están íntimamente relacionados.
- Los objetos tienen la propiedad de ocultar la información. Los objetos pueden saber cómo comunicarse entre sí a través de interfaces bien definidas, pero por lo general no se les permite conocer la manera en que se implementan los demás objetos.
- Los programadores en Java se concentran en la creación de sus propios tipos definidos por el usuario llamados clases. A los componentes de datos de las clases se les conoce como variables de instancia.
- Java utiliza la herencia para crear nuevas clases, a partir de las definiciones de clases existentes.
- Cada clase en Java es una subclase de **Object**. Entonces, cada nueva definición de una clase tiene los atributos (datos) y comportamientos (métodos) de la clase **Object**.

- Las palabras reservadas **`public`** y **`private`** son modificadores de acceso a los datos.
- Las variables de instancia y los métodos que se declaran con el modificador de acceso a datos **`public`** son accesibles en donde quiera que el programa haga referencia al objeto de la clase en la que están definidos.
- Las variables de instancia y los métodos que se declaran con el modificador de acceso a datos **`private`** son accesibles solamente en los métodos de la clase en la que están definidos.
- Por lo general, las variables de instancia se declaran **`private`** y, por lo general, los métodos se declaran **`public`**.
- Los clientes de una clase utilizan los métodos públicos (o servicios públicos) de dicha clase para manipular los datos almacenados en los objetos de la clase.
- Un constructor es un método con el mismo nombre que el de la clase que inicializa las variables de instancia de un objeto de la clase, cuando se crea la instancia de la misma clase. Los métodos constructores pueden sobrecargarse en una clase. Los constructores pueden tomar argumentos pero no pueden devolver valor alguno.
- Los constructores y otros métodos que modifican los valores de las variables de instancia siempre deben mantener a los objetos en un estado consistente.
- El método **`toString`** no toma argumentos y devuelve una **`String`** (Cadena). El método **`toString`** original de la clase **`Object`** es un contenedor que por lo general redefine una subclase.
- Cuando se crea la instancia de un objeto, el operador **`new`** reserva la memoria para ese objeto, luego **`new`** llama al constructor de la clase para inicializar las variables de instancia del objeto.
- Si los archivos **`.class`** para las clases utilizadas en un programa se encuentran en el mismo directorio de la clase que las utiliza, no se requieren instrucciones **`import`**.
- Concatenar un **`String`** y cualquier objeto provoca una llamada implícita al método **`toString`** del objeto para convertirlo en un **`String`**, luego se concatenan los **`Strings`**.
- Dentro del alcance de una clase, los miembros de la clase de inmediato están accesibles para todos los métodos de la clase y se puede hacer referencia a ellos simplemente por su nombre. Fuera del alcance de la clase, solamente se puede acceder a los miembros de la clase a través de un "manipulador" (es decir, una referencia a un objeto de la clase).
- Si un método define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase se oculta detrás de la variable con alcance de método dentro del mismo método. Se puede acceder a una variable de instancia oculta colocando antes del nombre la palabra reservada **`this`** y el operador punto.
- Cada clase e interfaz en el API de Java pertenece a un paquete específico que contiene un grupo de clases e interfaces relacionadas.
- En realidad, los paquetes son estructuras de directorios que se utilizan para organizar las clases y las interfaces. Los paquetes proporcionan un mecanismo para la reutilización de software y una convención para los nombres de clases úncos.
- Crear clases reutilizables requiere: definir una clase pública, agregar una instrucción **`package`** al archivo de definición de la clase, compilar la clase en la estructura de directorio apropiada para el paquete para tener la nueva clase disponible para el compilador y el intérprete, e importar la clase dentro de un programa.
- Java 2 tiene un directorio llamado **`classes`** en donde se coloca la versión compilada de algunas clases reutilizables que son bien conocidas tanto por el compilador como por el intérprete.
- Cuando compile una clase en un paquete, debe pasar la opción **`-d`** al compilador para especificar en dónde crear (o localizar) todos los directorios de la instrucción **`package`**.
- Los nombres del directorio **`package`** se vuelven parte del nombre de la clase cuando ésta se compila. Utilice este identificador completo en los programas, o importe las clases y utilice su nombre corto (el nombre de la clase por sí mismo) en el programa.
- Si no se definen constructores para una clase, el compilador crea un constructor predeterminado que no toma argumentos.
- Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos de la segunda clase.
- Las clases con frecuencia proporcionan métodos públicos para permitir a los clientes de la clase *establecer* (es decir, asignar valores) u *obtener* (es decir, adquirir valores) de variables de instancia privadas. Por lo general, los métodos *obtener* son conocidos como métodos de acceso o métodos de consulta. Por lo general, a los métodos *establecer* se les llama métodos mutantes (debido a que por lo general modifican un valor).
- Todo evento tiene una fuente; el componente GUI con el que el usuario interactúa para indicar al programa qué tarea realizar.
- Cuando no se proporciona ningún modificador de acceso a miembros para un método o una variable, cuando éste se define dentro de una clase, se considera que el método o la variable tienen acceso al paquete.

- Si un programa utiliza múltiples clases del mismo paquete, estas clases pueden acceder directamente a los métodos de acceso a paquetes y a los datos de los otros métodos, a través de una referencia a un objeto.
- Cada objeto tiene acceso a una referencia a sí mismo llamada referencia **`this`**, la cual puede utilizarse dentro de los métodos de la clase para hacer referencia explícita a los datos y objetos del objeto.
- En cualquier momento en el que usted tenga una referencia a un programa (incluso como resultado de una llamada a un método), la referencia puede ser seguida por un operador punto y una llamada a uno de los métodos del tipo de referencia.
- Todas las clases en Java pueden tener un método finalizador que devuelve los recursos al sistema. Un método finalizador de la clase siempre tiene el nombre **`finalize`**, no recibe parámetros y no devuelve valor alguno. El método **`finalize`** se define originalmente en la clase **`Object`** como un contenedor que no hace cosa alguna. Esto garantiza que cada clase contiene un método **`finalize`** para llamar al recolector de basura.
- Una variable estática de clase representa información para toda la clase; todos los objetos de la clase comparten la misma porción de información. Se puede acceder a los miembros públicos y estáticos de una clase a través de una referencia a cualquier objeto de dicha clase, o se puede acceder a ellos a través del nombre de la clase mediante el uso del operador punto.
- El método público y estático **`gc`** de la clase **`System`** sugiere que el colector de basura se ejecute inmediatamente. Esta sugerencia puede ignorarse. El recolector de basura no garantiza la recolección de todos los objetos en un orden específico.
- Un método declarado como **`static`** no tiene acceso a los miembros no estáticos de la clase. A diferencia de los métodos no estáticos, un método estático no contiene una referencia **`this`**, ya que las variables estáticas y los métodos estáticos de la clase existen independientemente de cualquier objeto de la clase.
- Los miembros estáticos de la clase existen, incluso si no existen objetos de dicha clase; éstos están disponibles tan pronto como se carga la clase en memoria en tiempo de ejecución.

## *TERMINOLOGÍA*

acceso a paquetes
alcance de una clase
atributo
biblioteca de clases
clase
clase contenedora
cliente de una clase
código reutilizable
comportamiento
constructor
constructor predeterminado
constructor sin argumentos
control de acceso a miembros
crear la instancia (instanciar) un objeto de una clase
definición de clase
encapsulamiento
estado consistente de una variable de instancia
extender
extensibilidad
finalizador
implementación de una clase
inicializar el objeto de una clase
instancia de una clase
instrucción **`package`**
interfaz de una clase
interfaz pública de una clase
llamadas a métodos
método
método de acceso
método de ayuda
método de consulta
método de instancia
método de una clase (**`static`**)
método de utilidad
método *establecer*
método mutante
método *obtener*
método predicado
método **`static`**
modificadores de acceso a miembros
objeto
ocultamiento de información
opción del compilador **`–d`**
operador **`new`**
operador punto (**`.`**)
principio del menor privilegio
private
programación basada en objetos (PBO)
programación orientada a objetos (POO)
public
referencia **`this`**
reutilización de software
servicios de una clase
tipo de dato
tipo de dato abstracto (ADT)
tipo definido por el programador
tipo definido por el usuario
variable de clase
variable de clase **`static`**
variable de instancia

## *ERRORES COMUNES DE PROGRAMACIÓN*

**26.1** Definir más de una clase pública en el mismo archivo, es un error de sintaxis.

**26.2** El hecho de que un método que no es un miembro de una clase en particular intente acceder a un miembro privado de dicha clase, es un error de sintaxis.

**26.3** Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor, es un error lógico. Java permite a otros métodos de la clase tener el mismo nombre de la clase y especificar los tipos de

retorno. Dichos métodos no son constructores y no se les llamará cuando se genere la instancia de un objeto de la clase.

**26.4** Si se proporcionan los constructores para la clase, pero ninguno de los constructores públicos es un constructor sin argumentos, y se intenta hacer una llamada al constructor sin argumentos para inicializar un objeto de la clase, ocurre un error de sintaxis. Es posible llamar a un constructor sin argumentos solamente si no existen constructores para esa clase (se llama al constructor predeterminado), o si no existe un constructor sin argumentos.

**26.5** Un constructor puede llamar a otros métodos de la clase, tal como los métodos *establecer* y *obtener*, pero debido a que el constructor inicializa el objeto, las variables de instancia no pueden aún estar en un estado consistente. El uso de las variables de instancia antes de inicializarse de manera apropiada, es un error.

**26.6** En un método en el que un parámetro del método tiene el mismo nombre que uno de los miembros de la clase, utilice explícitamente **`this`** si quiere tener acceso al miembro de la clase; de lo contrario, hará una referencia incorrecta al parámetro del método.

**26.7** Hacer referencia a **`this`** en un método **`static`**, es un error de sintaxis.

**26.8** Es un error de sintaxis que un método **`static`** llame a un método de instancia o que acceda a una variable de instancia.

## *BUENAS PRÁCTICAS DE PROGRAMACIÓN*

**26.1** Agrupe los miembros de acuerdo con los modificadores de acceso a miembros dentro de la definición de una clase, para mayor claridad y legibilidad.

**26.2** Nosotros preferimos listar primero a las variables de instancia **`private`** de una clase, para que conforme lea el código, vea los nombres y los tipos de dichas variables, antes de utilizarlas en los métodos de la clase.

**26.3** A pesar del hecho de que los miembros públicos y privados pueden repetirse y mezclarse, primero liste en un grupo a todos los miembros privados de la clase, y después liste en otro grupo a todos los miembros públicos.

**26.4** Siempre defina una clase de manera que sus variables de instancia se mantengan en un estado consistente.

**26.5** Inicialice las variables de instancia de una clase en el constructor de esa clase.

**26.6** Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicializa apropiadamente con valores significativos.

**26.7** Todo método que modifica las variables de instancia privadas de un objeto deben asegurarse de que los datos permanecen en un estado consistente.

**26.8** Evite utilizar nombres de parámetros que tengan conflictos con los nombres de los métodos de las clases.

**26.9** Siempre invoque métodos **`static`** por medio del nombre de la clase y del operador punto (**`.`**). Esto enfatiza a otros programadores que leen su código que el método que llaman es un método **`static`**.

## *TIPS DE RENDIMIENTO*

**26.1** Todos los objetos en Java se pasan por referencia. Sólo se pasa la dirección de memoria, no una copia de todo el objeto (como se haría en un paso por valor).

**26.2** Java conserva el almacenamiento, manteniendo sólo una copia de cada método por clase; este método es invocado por cada objeto de dicha clase. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase.

**26.3** Utilice las variables de clase **`static`** para ahorrar espacio, cuando sea suficiente una sola copia de los datos.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**26.1** Es importante escribir programas que sean claros y fáciles de mantener. La regla es el cambio, en lugar de la excepción. Los programadores deben prever que su código será modificado. Como veremos pronto, las clases facilitan la modificación de un programa.

**26.2** Las definiciones de las clases que comienzan con la palabra reservada **`public`** deben almacenarse en un archivo que tiene el mismo nombre que la clase, y terminar con la extensión de archivo **`.java`**.

**26.3** Toda clase definida en Java debe ser una extensión de otra clase. Si la clase no utiliza explícitamente la palabra reservada **`extends`** en su definición, esta clase implícitamente se extiende de **`Objects`**.

**26.4** Mantenga privadas todas las variables de instancia. Cuando sea necesario, proporcione métodos públicos para establecer los valores de variables de instancia privadas y para obtener los valores de variables de instancia privadas. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce los errores y mejora la posibilidad de modificación del programa.

**26.5** Los métodos tienden a caer en diversas categorías: métodos que obtienen los valores a partir de variables de instancia privadas; métodos que establecen los valores de las variables de instancia privadas; métodos que implementan los servicios de la clase; y métodos que realizan distintos mecanismos para la clase, tales como la inicialización de los objetos de las clases, la asignación de los objetos de las clases, y la conversión entre clases y los tipos predefinidos, o entre clases y otras clases.

**26.6** Cada vez que **`new`** crea un objeto de la clase, se llama al constructor de dicha clase para inicializar las variables de instancia del nuevo objeto.

**26.7** El ocultamiento de información promueve la capacidad de modificación del programa y simplifica la percepción de los clientes respecto a la clase.

**26.8** Los clientes de una clase pueden (y deben) utilizar la clase sin conocer los detalles de implementación de la clase. Si cambia la implementación de la clase (por ejemplo, para mejorar el rendimiento), la interfaz proporcionada permanece constante, el código fuente de los clientes de la clase no necesitan modificación. Esto hace mucho más fácil la modificación de los sistemas.

**26.9** Con frecuencia, utilizar un método de programación orientada a objetos simplifica las llamadas a los métodos, al reducir el número de parámetros a pasar. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de las variables de instancia y de los métodos dentro de un objeto le da a los métodos el derecho de acceso a las variables de instancia.

**26.10** Un archivo de código fuente en Java tiene el siguiente orden: una instrucción **`package`** (si existe alguna), instrucción **`import`** (si existen), y las definiciones de las clases. Solamente una de las definiciones de las clases puede ser pública. Las demás clases en el archivo también se colocan en el paquete, pero no son reutilizables. Éstas se encuentran en el paquete para soportar a la clase reutilizable del archivo.

**26.11** Los métodos que establecen los valores de datos privados deben verificar que los nuevos valores que se pretenden sean apropiados; si no lo son, los métodos establecer deben colocar las variables de instancia privadas en un estado consistente apropiado.

**26.12** Si un método de la clase proporciona toda o parte de la funcionalidad requerida por otro método de la clase, llame a dicho método desde otro método. Esto simplifica el mantenimiento del código y reduce la probabilidad de error si la implementación del código se modifica. También es un ejemplo claro de la reutilización.

**26.13** Acceder a los datos **`private`** a través de los métodos *establecer* y *obtener* no solamente protege a las variables de instancia de recibir valores no válidos, sino que además aísla a los clientes de la clase de la representación de las variables de instancia. Por lo tanto, si la representación de los datos cambia (por lo general, para reducir el almacenamiento requerido, o para mejorar el rendimiento), solamente necesita modificar las implementaciones del método; los clientes no necesitan modificación alguna mientras la interfaz proporcionada por los métodos permanezca igual.

**26.14** Cualquier variable de una clase **`static`** y cualquier método de una clase **`static`** puede utilizarse incluso si ningún objeto de esa clase se ha instanciado.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**26.1** Complete los espacios en blanco:

a) Se accede a los miembros de una clase a través del operador ____________, junto con una referencia a un objeto de la clase.
b) Se puede acceder a los miembros de una clase especificada como ____________ sólo por medio de métodos de la clase.
c) Un ____________ es un método especial que se utiliza para inicializar las variables de instancia de una clase.
d) Un método ____________ se utiliza para asignar valores a las variables de instancia privadas de una clase.
e) Los métodos de una clase normalmente se hacen ____________ y las variables de instancia de una clase normalmente se hacen ____________.
f) Un método ____________ se utiliza para recuperar los valores de datos privados de una clase.
g) La palabra reservada ____________ introduce la definición de una clase.
h) Los miembros de una clase especificados como ____________ están accesibles en cualquier parte en donde un objeto de la clase se encuentre en alcance.

i) El operador ____________ asigna de manera dinámica memoria para un objeto de un tipo especificado, y devuelve una ____________ para ese tipo.
j) Una variable de instancia ____________ representa información de toda la clase.
k) Un método declarado como **`static`** no puede acceder a los miembros ____________ de la clase.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**26.1** a) Punto (**`.`**). b) **`private`**. c) Constructor. d) Establecer. e) **`public`**, **`private`**. f) Obtener. g) **`class`**. h) **`public`**. i) **`new`**, referencia. j) **`static`**. k) No estáticos.

## EJERCICIOS

**26.2** Cree una clase llamada **`Racional`** para realizar operaciones aritméticas con fracciones. Escriba un programa controlador para probar su clase.

Utilice variables enteras para representar las variables de instancia privadas de la clase: el **`numerador`** y el **`denominador`**. Proporcione un método constructor que permita a un objeto de esta clase inicializarse cuando se declare. El constructor debe almacenar la fracción en forma reducida (es decir, la fracción

```
2/4
```

debe almacenarse en el objeto como 1 en el **`numerador`**, y 2 en el **`denominador`**). Proporcione un constructor sin argumentos que establezca valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos **`public`** para cada uno de los siguientes:

a) Suma de dos números racionales. El resultado de la suma debe almacenarse en forma reducida.
b) Resta de dos números racionales. El resultado de la resta debe almacenarse en forma reducida.
c) Multiplicación de dos números racionales. El resultado de la multiplicación debe almacenarse en forma reducida.
d) División de dos números racionales. El resultado de la división debe almacenarse en forma reducida.
e) Impresión de números racionales en la forma **`a/b`**, en donde **`a`** es el **`numerador`** y **`b`** es el **`denominador`**.
f) Impresión de números racionales en formato de punto flotante. (Considere el proporcionar capacidades de formato que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal.)

**26.3** Modifique la clase **`Hora2`** de la figura 26.3 para que incluya el método **`marcar`** que incremente en un segundo la hora almacenada en un objeto **`Hora2`**. También proporcione un método **`incrementaMinuto`** para incrementar los minutos, y el método **`incrementaHora`** para incrementar la hora. El objeto **`Hora2`** siempre debe permanecer en un estado consistente. Escriba un programa controlador que pruebe el método **`marcar`**, el método **`incrementaMinuto`** y el método **`incrementaHora`**, para garantizar que funcionan correctamente. Asegúrese de probar los siguientes casos:

a) Incrementar para llegar al siguiente minuto.
b) Incrementar para llegar a la siguiente hora,
c) Incrementar para llegar al día siguiente (es decir, 11:59:59 PM a 12:00:00 AM).

**26.4** Cree una clase **`Rectangulo`**. La clase tiene atributos **`longitud`** y **`ancho`**, cada uno con el valor predeterminado 1. Tiene métodos que calculan el **`perimetro`** y el **`area`** del rectángulo. Tiene métodos *establecer* y *obtener*, tanto para la **`longitud`** como para el **`ancho`**. Los métodos *establecer* deben verificar que la **`longitud`** y el **`ancho`** sean números de punto flotante mayores que 0.0 y menores que 20.0.

**26.5** Cree una clase **`Rectangulo`** más sofisticada que la que generó en el ejercicio anterior. Esta clase sólo almacena las coordenadas cartesianas de las cuatro esquinas del rectángulo. El constructor llama a un método *establecer* que acepta cuatro valores de coordenadas, y verifica que cada una de ellas se encuentre en el primer cuadrante y que ninguna *x* y *y* sea mayor que 20.0. El método *establecer* también verifica que las coordenadas proporcionadas, en realidad especifiquen un rectángulo. Proporcione métodos que calculen la **`longitud`**, el **`ancho`**, el **`perimetro`** y el **`area`**. La **`longitud`** es la más grande de las dos dimensiones. Incluya un método predicado **`esCuadrado`** que determine si el rectángulo es un cuadrado.

**26.6** Modifique la clase **`Rectangulo`** del ejercicio anterior para que incluya un método **`draw`** que despliegue el rectángulo dentro de un cuadro de 25 por 25 que encierre la parte del primer cuadrante en donde se encuentra el rectángulo. Utilice métodos de la clase **`Graphics`** para ayudar a que se despliegue el **`Rectangulo`**. Si se siente ambicioso, podría incluir métodos que escalen el tamaño del rectángulo, que lo roten y que lo muevan alrededor de la parte designada del primer cuadrante.

**26.7** Cree una clase **`EnteroEnorme`** que utilice un arreglo de dígitos de 40 elementos para que almacene enteros tan grandes como 40 dígitos cada uno. Proporcione métodos **`introduceEnteroEnorme`**, **`despliegaEnteroEnorme`**, **`sumaEnterosEnormes`** y **`restaEnterosEnormes`**. Para comparar objetos de **`EnteroEnorme`**, proporcione métodos **`esIgualQue`**, **`noEsIgualQue`**, **`esMayorQue`**, **`esMenorQue`**, **`esMayorOIgualQue`** y **`esMenorOIgualQue`**; cada uno de éstos es un método "predicado" que simplemente devuelve **`true`** si las relaciones se mantienen entre los dos **`EnteroEnorme`**, y devuelve **`false`** si la relación no se mantiene. Proporcione un método predicado **`esCero`**. Si se siente ambicioso, también proporcione el método **`multiplicaEnterosEnormes`**, el método **`divideEnterosEnormes`** y el método **`moduloDeEnterosEnormes`**.

**26.8** Cree la clase **`CuentaAhorro`**. Utilice una variable estática para almacenar la **`tasaInteresAnual`** para todas las cuentas de ahorros. Cada objeto de la clase contiene una variable de instancia privada **`saldoAhorro`** que indica el monto que el ahorrador tiene en depósito. Proporcione el método **`calculaInteresMensual`**, el cual multiplica **`saldoAhorro`** por **`tasaInteresAnual`** dividida entre 12. Este interés debe sumarse a **`saldoAhorro`**. Proporcione un método estático **`modificaTasaInteres`** que establezca un nuevo valor para **`tasaInteresAnual`**. Escriba un programa para probar **`CuentaAhorro`**. Cree dos instancias para los objetos **`CuentaAhorro`**, **`ahorrador1`** y **`ahorrador2`**, con saldos de $2000.00 y $3000.00 respectivamente. Establezca **`tasaInteresAnual`** en 4%, luego calcule el interés mensual e imprima los nuevos saldos para cada cuenta. Posteriormente establezca **`tasaInteresAnual`** en 5% y calcule el interés del siguiente mes e imprima los nuevos saldos para cada cuenta.

**26.9** Cree la clase **`EstebleceEntero`**. Cada objeto de la clase puede almacenar enteros en el rango de 0 a 100. Un conjunto está representado internamente por un arreglo de valores booleanos. El elemento **`a[i]`** del arreglo es **`true`** (verdadero) si el entero *i* se encuentra en el conjunto. El elemento **`a[j]`** es **`false`** si el entero *j* no se encuentra en el conjunto. El constructor sin argumentos inicializa un conjunto llamado "conjunto vacío" (es decir, un conjunto cuya representación de arreglo contiene solamente valores **`false`**).

Proporcione los siguientes métodos: el método **`unionConjuntosEnteros`** crea un tercer conjunto que es la unión teórica de los dos conjuntos existentes (es decir, un elemento del tercer arreglo o conjunto se establece en **`true`** si dicho elemento es **`true`** en uno o en los dos conjuntos existentes; de lo contrario, el elemento del tercer conjunto se establece en **`false`**). El método **`interseccionConjuntosEnteros`** crea un tercer conjunto que es la intersección teórica de los dos conjuntos existentes, es decir, un elemento del tercer conjunto o arreglo se establece en **`false`** si dicho elemento es **`false`** en uno o en los dos conjuntos existentes; de lo contrario el elemento del tercer conjunto se establece en **`true`**). El método **`insertaElemento`** inserta un nuevo entero **`k`** dentro de un conjunto (al establecer **`a[k]`** a **`true`**). El método **`eliminaElemento`** elimina el entero *m* (al establecer **`a[m]`** en **`false`**). El método **`estableceImpresion`** imprime un conjunto como una lista de números separada por espacios. Imprime solamente los elementos que están presentes en el conjunto. Imprime —- para un conjunto vacío. El método **`esIgualQue`** determina si dos conjuntos son iguales. Escriba un programa para probar su clase **`ConjuntoEnteros`**. Cree varias instancias de objetos **`ConjuntoEnteros`**. Pruebe que todos los métodos funcionan apropiadamente.

**26.10** Sería perfectamente razonable para la clase **`Hora1`** de la figura 26.1 representar la hora internamente como el número de segundos desde la medianoche, en lugar de los tres valores enteros para la hora, los minutos y los segundos. Los clientes podrían utilizar los mismos métodos públicos y obtener los mismos resultados. Modifique la clase **`Hora1`** de la figura 26.1 para implementar **`Hora1`** como el número de segundos desde medianoche y mostrar que no existe un cambio visible para los clientes de la clase.

**26.11** (*Programa de dibujo.*) Cree un applet de dibujo que dibuje líneas, rectángulos y elipses al azar. Para este propósito, cree un conjunto de clase de formas "inteligentes" en donde los objetos de esta clases sepan cómo dibujarse a sí mismas si se les proporciona un objeto **`Graphics`** que les diga en dónde dibujarse (es decir, el objeto **`Graphics`** del applet permite a una forma dibujar en el fondo del applet). Los nombres de clases debe ser **`MiLinea`**, **`MiRecta`** y **`MiElipse`**.

Los datos de la clase **`MiLinea`** deben incluir las coordenadas *x1*, *y1*, *x2* e *y2*. El método **`drawLine`** de la clase **`Graphics`** conectará mediante una línea los dos puntos proporcionados. Los datos de las clases **`MiRecta`** y **`MiElipse`** deben incluir el valor *x* de la coordenada superior izquierda, el valor *y* de la coordenada superior izquierda, un *ancho* (debe ser positivo) y una *altura* (debe ser positiva). Todos los datos de cada clase deben ser privados.

Además de los datos, cada clase debe definir al menos los siguientes métodos públicos:

a) Un constructor sin argumentos que establezca las coordenadas en 0.

b) Un constructor con argumentos que establezca las coordenadas con los valores proporcionados.

c) Métodos *establecer* para cada figura individual, que permita al programador establecer cada pieza de dato en la figura (por ejemplo, si usted tiene una variable de instancia **`x1`**, debe tener un método **`estableceX1`**).

d) Los métodos *obtener* para cada pieza de datos individual, que permitan al programador recuperar de manera independiente cada pieza de datos de la figura (por ejemplo, si usted tiene una variable de instancia **x1**, debe tener un método **obtieneX1**).
e) Un método **draw** con la primera línea

```
public void draw( Graphics g )
```

será llamado desde el método **paint** del applet para dibujar una figura en la pantalla.

Los métodos anteriores son indispensables. Si usted desea proporcionar más métodos para mayor flexibilidad, hágalo.

Comience con la definición de la clase **MiLinea** y un applet para probar sus clases. El applet debe tener una variable de instancia línea de **MiLinea** que pueda hacer referencia a un objeto **MiLinea** (creado en el método **init** del applet con coordenadas al azar). El método **paint** del applet debe dibujar la figura con una instrucción como

```
linea.draw( g );
```

en donde **linea** es una referencia a **MiLinea** y **g** es el objeto de **Graphics** que la forma utilizará para dibujarse a sí misma en el applet.

Después, modifique la referencia individual a **MiLinea** dentro de un arreglo de referencias a **MiLinea** y copie el código para varios objetos **MiLinea** dentro del programa de dibujo. El método **paint** del applet debe recorrer el arreglo de objetos **MiLinea** y dibujar cada uno.

Una vez que la parte anterior ya funcione, debe definir las clases **MiElipse** y **MiRecta**, y agregar los objetos de estas clases a los arreglos **MiRecta** y **MiElipse**. El método **paint** del **applet** debe recorrer cada arreglo y dibujar cada figura. Cree cinco figuras de cada tipo.

Una vez que el applet funcione, seleccione **Volver a cargar** del menú **Subprograma** para volver a cargar el applet. Esto provocará que el applet elija nuevos números al azar para dibujar las figuras.

En el capítulo 27, modificaremos este ejercicio para aprovechar las similitudes entre las clases, y así evitar el reinventar la rueda.

# 27

# Programación orientada a objetos en Java

## Objetivos

- Comprender la herencia y la reutilización de software.
- Comprender las superclases y las subclases.
- Apreciar cómo es que el polimorfismo hace que los sistemas sean extensibles y que se puedan mantener.
- Comprender la diferencia entre clases abstractas y clases concretas.
- Aprender cómo crear clases abstractas (**`abstract`**) e interfaces.

*No digas que conoces completamente a alguien, hasta que hayas compartido una herencia con él.*
Johann Kasper Lavater

*Este método es para definir como el número de una clase a la clase de todas las clases similares a la clase dada.*
Bertrand Russell

*Es bueno heredar una biblioteca, pero es mejor formar una.*
Augustine Birrell

*Las proposiciones generales no deciden casos concretos.*
Oliver Wendell Holmes

*Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, hasta cierto punto, condicionadas por lo que se sabe, o no se sabe, en la época en la que vive.*
Alfred North Whitehead



**Plan general**



## 27.1 Introducción

En este capítulo explicamos la programación orientada a objetos (POO) y sus tecnologías componentes clave: la *herencia* y el *polimorfismo*. La herencia es una forma de reutilización de software, en la que se crean nuevas clases a partir de clases existentes, absorbiendo sus atributos y sus comportamientos y mejorándolas con capacidades que requieren las nuevas clases. La reutilización de software ahorra tiempo en el desarrollo de programas, lo cual motiva el uso de software de alta calidad probado y depurado, con lo que se reducen los problemas que se generan cuando un sistema empieza a utilizarse. Éstas son posibilidades excitantes. El polimorfismo nos permite escribir programas de modo general para manejar una amplia variedad de clases relacionadas existentes. El polimorfismo facilita el agregar nuevas capacidades a un sistema. La herencia y el polimorfismo son técnicas efectivas para lidiar con la complejidad del software.

Cuando el programador crea una nueva clase, en lugar de escribir variables y métodos de instancia completamente nuevos, puede designar que la nueva clase herede las variables y los métodos de instancia de una *superclase* previamente definida. A la nueva clase se le conoce como una *subclase*. Cada subclase por sí misma se vuelve una candidata para ser una superclase para algunas subclases futuras.

La *superclase directa* de una subclase es la superclase de la que la subclase directamente hereda (vía la palabra reservada `extends`). Una superclase indirecta hereda desde dos o más niveles superiores en la jerarquía de clase.

Por medio de la *herencia simple*, una clase se deriva de una superclase. Java no soporta la *herencia múltiple* (como C++ lo hace), pero sí soporta la idea de las *interfaces*. Las interfaces ayudan a Java a tener muchas de las ventajas de la herencia múltiple sin los problemas asociados. En este capítulo explicaremos los detalles de las interfaces; consideraremos los principios generales y un ejemplo detallado sobre la creación y el uso de las interfaces.

Una subclase normalmente agrega por su cuenta variables y métodos de instancia, por lo que una subclase generalmente es más grande que su superclase. Una subclase es más específica que su superclase y representa un grupo más pequeño de objetos. Con la herencia simple, la subclase inicia prácticamente igual que la superclase. La fortaleza real de la herencia proviene de la habilidad de definir en la subclase agregados, o reemplazos, para las características heredadas de la superclase.

Todo objeto de una subclase es también un objeto de la superclase de esa subclase. Sin embargo, lo inverso no es verdad; los objetos de una superclase no son objetos de las subclases de esa superclase. Nosotros aprovecharemos la relación "el objeto de una subclase es un objeto de la superclase" para realizar algunas manipulaciones poderosas. Por ejemplo, por medio de la herencia podemos vincular una amplia variedad de objetos diferentes, relacionados con una superclase común, en una lista ligada de objetos de una superclase. Esto permite que una variedad de objetos se procesen en una manera general. Como veremos en este capítulo, es la idea central de la programación orientada a objetos.

En este capítulo agregamos una nueva forma de control de acceso a miembros, a saber, el acceso **`protected`** (protegido). Los métodos de una subclase y los métodos de otras clases en el mismo paquete de la superclase pueden acceder a los miembros **`protected`** de la superclase.

La experiencia en construir sistemas de software indica que partes importantes de código lidian con casos especiales muy relacionados. En tales sistemas se torna difícil ver la "imagen completa", ya que el diseñador y el programador se preocupan por los casos especiales. La programación orientada a objetos proporciona diversas formas para "ver el bosque a través de los árboles"; un proceso llamado *abstracción*.

Si un programa por procedimientos tiene muchos casos especiales muy relacionados, entonces es común ver estructuras **`switch`** o estructuras **`if/else`** anidadas que diferencian los casos especiales y proporcionan la lógica de procesamiento para manejar individualmente cada caso. Mostraremos cómo utilizar la herencia y el polimorfismo para remplazar dicha lógica de **`switch`** con una lógica mucho más sencilla.

Plantearemos la diferencia entre la *relación es un* y la *relación tiene un*. *Es un* es herencia. En una relación *es un*, un objeto de un tipo correspondiente a una subclase también puede tratarse como un objeto de un tipo de su superclase. *Tiene un* es composición (como explicamos en el capítulo 26). En una relación *tiene un*, un objeto de una clase tiene como miembros a uno o más objetos de otras clases. Por ejemplo, un automóvil *tiene un* volante.

Los métodos de una subclase podrían necesitar acceder a ciertas variables y ciertos métodos de instancia de su superclase.



**Observación de ingeniería de software 27.1**

*Una subclase no puede acceder directamente a miembros **`private`** de su superclase.*

Éste es un aspecto crucial de la ingeniería de software en Java. Si una subclase pudiera acceder a los miembros **`private`** de una superclase, se violaría el ocultamiento de información en la superclase.



**Tip para prevenir errores 27.1**

*Ocultar los miembros **`private`** es una gran ayuda al probar, depurar y modificar correctamente los sistemas. Si una subclase pudiera acceder a los miembros **`private`** de su superclase, entonces sería posible que las clases derivadas de esa subclase accedieran también a esos datos, y así sucesivamente. Esto propagaría el acceso a lo que se supone deberían ser datos **`private`**, y los beneficios del ocultamiento de información se perderían a lo largo de la jerarquía de la clase.*

Una subclase en el mismo paquete de su superclase puede acceder a los miembros **`public`**, **`protected`** y miembros de acceso al paquete de su superclase. Los miembros de una superclase que no deben acceder a una subclase por medio de la herencia, se declaran como **`private`** en la superclase. Una subclase puede efectuar modificaciones de estado a los miembros **`private`** de una superclase, sólo a través de métodos **`public`**, **`protected`** y de acceso a paquetes provistos en la superclase y heredados a la subclase.

Un problema con la herencia es que una subclase puede heredar métodos que no necesita, o que no debe tener. Cuando un miembro de una superclase es inadecuado para una subclase, ese miembro puede *redefinirse* en la subclase con una implementación adecuada.

Tal vez lo más excitante sea la noción de que las nuevas clases pueden ser herederas de muchas *bibliotecas de clases*. Las empresas desarrollan sus propias bibliotecas de clases y aprovechan otras disponibles alrededor del mundo. Algún día, la mayoría del software se construirá a partir de *componentes reutilizables estandarizados*,

tal como se construye actualmente la mayoría del hardware. Esto ayudará a cumplir con el reto de desarrollar software poderoso que necesitaremos en el futuro.

## 27.2 Superclases y subclases

Con frecuencia, un objeto de una clase también *es un* objeto de otra clase. Un rectángulo ciertamente *es un* cuadrilátero (como los cuadrados, los paralelogramos y los trapezoides). Entonces, puede decirse que la clase **`Rectangulo`** hereda de la clase **`Cuadrilatero`**. En este contexto, la clase **`Cuadrilatero`** es una superclase y la clase **`Rectangulo`** es una subclase. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto afirmar que un cuadrilátero *es un* rectángulo (el cuadrilátero podría ser un paralelogramo). La figura 27.1 muestra diversos ejemplos de herencia simple de superclases y subclases potenciales.

La herencia normalmente produce subclases con *más* características que sus superclases, por lo que los términos superclase y subclase pueden ser confusos. Sin embargo, existe otra manera de ver estos términos, la cual hace clara la relación. Todo objeto de una subclase *es un* objeto de su superclase, y una superclase puede tener muchas subclases, por lo que el conjunto de objetos representados por una superclase normalmente es más grande que el conjunto de objetos representados por cualquier subclase de esa superclase. Por ejemplo, la superclase **`Vehiculo`** representa de manera general a todos los vehículos, como automóviles, camiones, botes, bicicletas, etcétera. Sin embargo, la subclase **`Automovil`** representa sólo a un pequeño subconjunto de todos los vehículos en el mundo.

Las relaciones de herencia forman estructuras jerárquicas parecidas a un árbol. Una superclase existe en una relación jerárquica con sus subclases. Ciertamente, una clase puede existir por sí misma, pero es cuando una clase se utiliza con el mecanismo de la herencia, que la clase se vuelve una superclase que proporciona atributos y comportamientos a otras clases, o que la clase se vuelve una subclase que hereda dichos atributos y comportamientos.

Desarrollemos una jerarquía de herencia simple para figuras. Los círculos, cuadrados, cubos y tetraedros son diferentes tipos de figuras. Algunas de estas figuras pueden dibujarse en dos dimensiones, y algunas otras deben modelarse en tres dimensiones. Esto arroja la jerarquía de herencia que aparece en la figura 27.2. Observe que esta jerarquía podría contener muchas otras clases. Por ejemplo, los cuadrados y los rectángulos son cuadriláteros. Las flechas en la jerarquía representan la relación *es un*. Por ejemplo, basándonos en esta jerarquía de clase podemos decir que "un **`Cuadrado`** *es una* **`FiguraBidimensional`**", o que "un **`Cubo`** *es una* **`FiguraTridimensional`**". **`Figura`** es la *superclase directa* tanto de **`FiguraBidimensional`** como de **`FiguraTridimensional`**. **`Figura`** es una *superclase indirecta* de todas las demás clases del diagrama de jerarquía.

| Superclase | Subclases |
|---|---|
| **`Estudiante`** | **`EstudianteTitulado`** |
| | **`EstudianteUniversitario`** |
| **`Figura`** | **`Circulo`** |
| | **`Triangulo`** |
| | **`Rectangulo`** |
| **`Prestamo`** | **`PrestamoAutomotriz`** |
| | **`PrestamoMejorarCasa`** |
| | **`PrestamoHipotecario`** |
| **`Empleado`** | **`EmpleadoDocente`** |
| | **`EmpleadoAdministrativo`** |
| **`Cuenta`** | **`CuentaCheques`** |
| | **`CuentaAhorros`** |

**Figura 27.1** Algunos ejemplos de herencia simple.

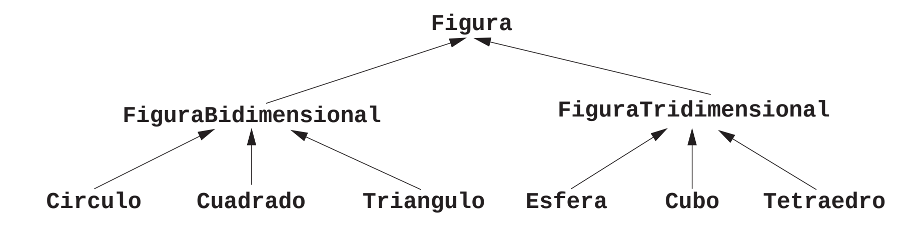



**Figura 27.2** Una parte de la jerarquía de la clase **`Figura`**.

Además, si partimos de la parte inferior del diagrama, podemos seguir las flechas y aplicar la relación *es un* hacia arriba, hasta llegar a la parte superior de la jerarquía de la superclase. Por ejemplo, un **`Tetraedro`** *es una* **`FiguraTridimensional`** y también *es una* **`Figura`**. En Java, un **`Tetraedro`** también es un **`Object`**, ya que todas las clases en Java tienen a **`Object`** como una de sus superclases directas o indirectas. Por lo tanto, todas las clases en Java tienen una relación jerárquica en la que comparten los 11 métodos definidos por la clase **`Object`**, la cual incluye los métodos **`toString`** y **`finalize`** que explicamos anteriormente. Explicaremos otros métodos de la clase **`Object`** conforme los necesitemos en el texto.

En el mundo existen muchos ejemplos de jerarquías, pero los estudiantes no están acostumbrados a categorizar al mundo de esta manera, por lo que son necesarios ciertos ajustes a su pensamiento. De hecho, los estudiantes de biología han tenido prácticas con jerarquías. Todo lo que estudiamos en biología está agrupado en una jerarquía encabezada por los seres vivos, los cuales pueden ser plantas, animales, etcétera.

Para especificar que la clase **`FiguraBidimensional`** se deriva (o hereda de) la clase **`Figura`**, la clase **`FiguraBidimensional`** podría definirse en Java como:

```
class FiguraBidimensional extends Figura { ... }
```

Con la herencia, los miembros **`private`** de una superclase no son directamente accesibles para las subclases de esa clase. Los miembros de acceso al paquete de la superclase sólo son accesibles en una subclase, si la superclase y su subclase están en el mismo paquete. Todos los demás miembros de la superclase se vuelven miembros de la subclase, utilizando su acceso a miembros original (es decir, los miembros **`public`** de la superclase se vuelven miembros **`public`** de la subclase, y los miembros **`protected`** de la superclase se vuelven miembros **`protected`** de la subclase).



**Observación de ingeniería de software 27.2**

*Los constructores nunca se heredan; éstos son específicos de la clase en la que están definidos.*

Es posible tratar a los objetos de una superclase y a los objetos de una subclase de manera similar; esa similitud se expresa en los atributos y comportamientos de la superclase. Los objetos de todas las clases derivadas de una superclase común pueden tratarse como objetos de esa superclase.

Consideraremos muchos ejemplos en los que aprovecharemos esta relación con una programación sencilla no disponible en lenguajes no orientados a objetos como C.

## 27.3 Miembros `protected`

Los miembros **`public`** de una superclase son accesibles desde cualquier parte del programa que tenga una referencia hacia ese tipo de superclase o hacia uno de los tipos de su subclase. Los miembros **`private`** de una superclase sólo son accesibles en los métodos de esa superclase.

Los miembros **`protected`** de una superclase sirven como un nivel intermedio de protección entre el acceso **`public`** y el **`private`**. Se puede acceder a los miembros **`protected`** de una superclase sólo por medio de métodos de la superclase, por medio de métodos de subclases y por medio de métodos de otras clases en el mismo paquete (los miembros **`protected`** tienen acceso a paquetes).

Los métodos de subclases normalmente pueden hacer referencia a miembros **`public`** y **`protected`** de la superclase, simplemente utilizando los nombres de los miembros. Cuando un método de una subclase *redefine* un método de una superclase (explicado en la sección 27.4), se puede acceder al método de la superclase

desde la subclase, precediendo el nombre del método de la superclase con la palabra reservada **`super`**, seguida por el operador punto (`.`). Ilustramos esta técnica varias veces a lo largo del capítulo.

## 27.4 Relación entre objetos de superclases y objetos de subclases

Un objeto de una subclase puede tratarse como un objeto de su superclase. Esto hace posible realizar algunas manipulaciones interesantes. Por ejemplo, a pesar del hecho de que los objetos de una variedad de clases derivadas de una superclase en particular pueden ser muy diferentes entre sí, podemos crear un arreglo de referencias hacia ellos; siempre y cuando los tratemos como objetos de una superclase. Sin embargo, lo contrario no es verdad: un objeto de una superclase no es automáticamente un objeto de una subclase.



**Error común de programación 27.1**

*Tratar a un objeto de una superclase como un objeto de una subclase puede ocasionar errores.*

Sin embargo, se puede utilizar una conversión de tipo explícita para convertir una referencia de una superclase en una referencia de una subclase. Esto únicamente puede hacerse cuado la referencia de la superclase en realidad hace referencia a un objeto de una subclase; de lo contrario, Java indica una ***`ClassCastException`***; una indicación de que la operación de conversión de tipo no está permitida.



**Error común de programación 27.2**

*Asignar un objeto de una superclase a una referencia de una subclase (sin una conversión de tipo), es un error de sintaxis.*



**Observación de ingeniería de software 27.3**

*Si un objeto se ha asignado a una referencia de una de sus superclases, es aceptable convertir el tipo de ese objeto de regreso a su propio tipo. De hecho, esto debe hacerse para enviar a ese objeto cualquiera de los mensajes que no aparecen en esa superclase.*

Nuestro primer ejemplo de herencia aparece en la figura 27.3. Todos los applets que definimos han utilizado alguna de las técnicas que presentamos aquí. Ahora formalizaremos el concepto de la herencia.

```
// Figura 27.3: Punto.java
// Definición de la clase Punto

public class Punto {
   protected int x, y; // coordenadas del Punto

   // Constructor sin argumentos
   public Punto()
   {
      // llamada implícita al constructor de la superclase
      establecePunto( 0, 0 );
   } // fin del constructor Punto

   // Constructor
   public Punto( int a, int b )
   {
      // llamada implícita al constructor de la superclase
      establecePunto( a, b );
   } // fin del constructor Punto

   // Establece las coordenadas x y y del Punto
   public void establecePunto( int a, int b )
   {
```

**Figura 27.3** Asignación de referencias de subclases a referencias de superclases; **`Punto.java`**. (Parte 1 de 2.)

```
      x = a;
      y = b;
   } // fin del método establecePunto

   // obtiene la coordenada x
   public int obtieneX() { return x; }

   // obtiene la coordenada y
   public int obtieneY() { return y; }

   // convierte el punto a una representación como String representation
   public String toString()
      { return “[“ + x + “, “ + y + “]”; }
} // fin de la clase Punto
```

**Figura 27.3** Asignación de referencias de subclases a referencias de superclases; **`Punto.java`**. (Parte 2 de 2.)

```
// Figura 27.3: Circulo.java
// Definición de la clase Circulo

public class Circulo extends Punto {  // hereda desde punto
   protected double radio;

   // Constructor sin argumentos
   public Circulo()
   {
      // llamada implícita al constructor de la superclase
      estableceRadio( 0 );
   } // fin del constructor Circulo

   // Constructor
   public Circulo( double r, int a, int b )
   {
      super( a, b );  // llama al constructor de la superclase
      estableceRadio( r );
   } // fin del constructor Circulo

   // Establece el radio de Circulo
   public void estableceRadio( double r )
      { radio = ( r >= 0.0 ? r : 0.0 ); }

   // Obtiene el radio de Circulo
   public double obtieneRadio() { return radio; }

   // Calcula el área de Circulo
   public double area() { return Math.PI * radio * radio; }

   // convierte el Circulo a String
   public String toString()
   {
      return “Centro = “ + “[“ + x + “, “ + y + “]” +
             “; Radio = “ + radio;
   } // fin del método toString
} // fin de la clase Circulo
```

**Figura 27.3** Asignación de referencias de subclases a referencias de superclases; **`Circulo.java`**.

```
// Figura 27.3: PruebaHerencia.java
// Demostración de la relación “es un”
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class PruebaHerencia {
   public static void main( String args[] )
   {
      Punto refPunto, p;
      Circulo refCirculo, c;
      String salida;

      p = new Punto( 30, 50 );
      c = new Circulo( 2.7, 120, 89 );

      salida = “Punto p: “ + p.toString() +
               “\nCirculo c: “ + c.toString();

      // utiliza la relación “es un” para hacer referencia a Circulo
      // mediante una referencia a Punto
      refPunto = c;   // asigna Circulo a refPunto

      salida += “\n\nCirculo c (via refPunto): “ +
                 refPunto.toString();

      // Utiliza la conversión hacia abajo (convierte una referencia a una
      // superclase a un tipo de dato subclase) para asignar refPunto a
         refCirculo
      refCirculo = (Circulo) refPunto;

      salida += “\n\nCirculo c (mediante refCirculo): “ +
                 refCirculo.toString();

      DecimalFormat precision2 = new DecimalFormat( “0.00” );
      salida += “\nArea de c (via refCirculo): “ +
                 precision2.format( refCirculo.area() );

      // Intenta hacer referencia a un objeto Punto
      // mediante la referencia a Circulo
      if ( p instanceof Circulo ) {
         refCirculo = (Circulo) p;  // línea 40 en Prueba.java
         salida += “\n\nconversión exitosa”;
      }
      else
         salida += “\n\np no hace referencia a Circulo”;

      JOptionPane.showMessageDialog( null, salida,
         “Demuestra la \”relación \” es un”,
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin de main
} // fin de la clase PruebaHerencia
```

**Figura 27.3** Asignación de referencias de subclases a referencias de superclases; **`PruebaHerencia.java`**. (Parte 1 de 2.)

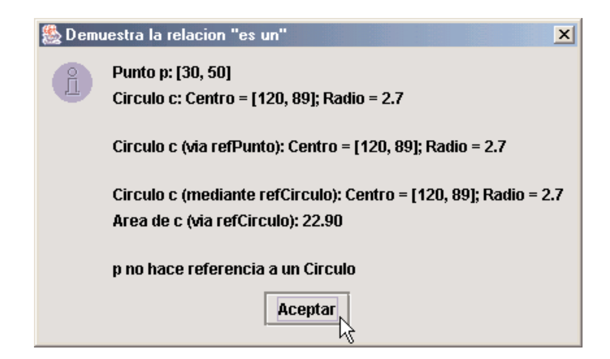



**Figura 27.3** Asignación de referencias de subclases a referencias de superclases; **`PruebaHerencia.java`**. (Parte 2 de 2.)

En Java, toda definición de clase debe extender a otra clase. Sin embargo, observe que la clase **`Punto`** (línea 4) no utiliza explícitamente la palabra reservada **`extends`**. Si la definición de una nueva clase no extiende explícitamente una definición de clase existente, Java utiliza implícitamente la clase **`Object`** (paquete **`java.lang`**) como la superclase para la definición de la nueva clase. La clase **`Object`** proporciona un conjunto de métodos que pueden utilizarse con cualquier objeto de cualquier clase.



### Observación de ingeniería de software 27.4

*Toda clase en Java extiende a* **`Object`**, *a menos que se especifique lo contrario en la primera línea de la definición de la clase. Por lo tanto, la clase* **`Object`** *es la superclase de toda la jerarquía de clases de Java.*

Las líneas 1 a 37 muestran la definición de la clase **`Punto`**. Las líneas 38 a 74 muestran la definición de la clase **`Circulo`**; veremos que la clase **`Circulo`** hereda de la clase **`Punto`**. Las líneas 75 a 126 muestran una aplicación que demuestra la asignación de referencias de subclase a referencias de superclase, y la conversión de tipo de referencias de superclase a referencias de subclase.

Primero examinemos la definición de la clase **`Punto`** de las líneas 1 a 37. Los servicios **`public`** de la clase **`Punto`** incluye los métodos **`establecePunto`**, **`obtieneX`**, **`obtieneY`**, **`toString`** y dos constructores **`Punto`**. Las variables de instancia **`x`** y **`y`** de **`Punto`** se especifican como **`protected`**. Esto evita que los clientes de objetos **`Punto`** accedan directamente a los datos (a menos que sean clases del mismo paquete), pero permite a las clases derivadas de **`Punto`** acceder directamente a las variables de instancia heredadas. Si los datos se especificaran como **`private`**, los métodos no privados de **`Punto`** tendrían que utilizarse para acceder a los datos, incluso las subclases. Observe que el método **`toString`** de **`Punto`** redefine el método **`toString`** original de la clase **`Object`**.

Los constructores de la clase **`Punto`** (líneas 8 y 15) deben llamar al constructor de la clase **`Object`**. De hecho, todo constructor de subclases es necesario para llamar al constructor de su superclase directa, ya sea implícita o explícitamente, como su primera tarea (por el momento, la sintaxis de esta llamada se explica con la clase **`Circulo`**). Si no hay una llamada explícita al constructor de la superclase, Java intenta llamar al constructor predeterminado de la superclase. Observe que las líneas 10 y 17 son comentarios que indican en dónde ocurre la llamada al constructor predeterminado de la superclase **`Object`**.

La clase **`Circulo`** (líneas 38 a 74) hereda de la clase **`Punto`**. Esto se especifica en la primera línea de la definición de la clase

```
public class Circulo extends Punto { // hereda de Punto
```

La palabra reservada **`extends`** en la definición de la clase indica la herencia. Todos los miembros (no privados) de la clase **`Punto`** (excepto los constructores) se heredan a la clase **`Circulo`**. Por lo tanto, la interfaz **`public`** de **`Circulo`** incluye los métodos **`public`** de **`Punto`**, así como los dos constructores sobrecargados de **`Circulo`** y los métodos de **`Circulo`** **`estableceRadio`**, **`obtieneRadio`**, **`area`** y **`toString`**. Observe que el método **`area`** (línea 66) utiliza la constante predefinida **`Math.PI`** de la clase **`Math`** (paquete **`java.lang`**) para calcular el área de un círculo.

Los constructores de **Circulo** (líneas 45 y 52) deben invocar un constructor **Punto** para inicializar la parte de superclase (variables **x** y **y** heredadas de **Punto**) de un objeto **Circulo**. El constructor predeterminado de la línea 45 no llama explícitamente a un constructor **Punto**, por lo que Java llama al constructor predeterminado de **Punto** (definido en la línea 8), el cual inicializa en ceros a los miembros de la superclase **x** y **y**. Si la clase **Punto** contuviera sólo el constructor de la línea 15 (es decir, no proporcionara un constructor predeterminado), ocurriría un error de compilación.

La línea 54 del cuerpo del segundo constructor **Circulo**

```
super( a, b ); // llamada explícita al constructor de la  superclase
```

invoca explícitamente al constructor **Punto** (definido en la línea 11) por medio de la sintaxis de llamada al constructor de la superclase [es decir, la palabra reservada ***super***, seguida por un conjunto de paréntesis que contienen los argumentos del constructor de la superclase (en este caso los valores **a** y **b** son pasados para inicializar a los miembros de la superclase **x** y **y**)]. La llamada al constructor de la superclase debe ser la primera línea del cuerpo del constructor de la subclase. Para llamar explícitamente al constructor predeterminado de la superclase, utilice la instrucción

```
super(); // llamada explícita al constructor predeterminado de la superclase
```



### Error común de programación 27.3

*Si una subclase hace una llamada **super** al constructor de su superclase, y esta llamada no es la primera instrucción en el constructor de la subclase, es un error de sintaxis.*



### Error común de programación 27.4

*Si los argumentos de una llamada **super** de una subclase al constructor de su superclase no coinciden con los parámetros especificados en una de las definiciones del constructor de la superclase, es un error de sintaxis.*

Una subclase puede redefinir el método de una superclase, utilizando la misma firma; a esto se le conoce como *redefinir* un método de superclase. Siempre que se menciona a un método por su nombre en la subclase, se llama a la versión de la subclase. De hecho, hemos redefinido métodos en todos los applets del libro. Cuando extendemos **JApplet** para crear una nueva clase applet, la nueva clase hereda las versiones de **init** y **paint** (y muchos otros métodos). Cada vez que definimos **init** o **paint**, redefinimos la versión original que se heredó. Además, cuando proporcionamos el método **toString** para las clases del capítulo 26, redefinimos la versión original de **toString** provista por la clase **Object**. Como veremos pronto, la referencia **super**, seguida por el operador punto, puede utilizarse para acceder a la versión original de la superclase de ese método desde la subclase.

Observe que el método **toString** de la clase **Circulo** (línea 69) redefine el método **toString** de la clase **Punto** (línea 35). El método **toString** de la clase **Punto** redefine el método original **toString** provisto por la clase **Object**. La clase **Object** proporciona el método original **toString**, por lo que todas las clases heredan un método **toString**. Este método se utiliza para convertir cualquier objeto de cualquier clase en una representación **String** y algunas veces es llamado implícitamente por el programa (por ejemplo, cuando se agrega un objeto a una **String**). El método **toString** de **Circulo** accede directamente a las variables de instancia **protected** **x** y **y** que se heredaron de la clase **Punto**. Los valores **x** y **y** se utilizan como parte de la representación **String** de **Circulo**. De hecho, si estudia el método **toString** de **Punto** y el método **toString** de la clase **Circulo**, notará que **toString** de **Circulo** utiliza exactamente el mismo formato que **toString** de **Punto** para las partes **Punto** del **Circulo**. Para llamar a **toString** de **Punto** desde la clase **Circulo**, utilice la expresión

```
super.toString()
```



### Observación de ingeniería de software 27.5

*Una redefinición de un método de una superclase en una subclase no tiene la misma firma que el método de la superclase. Tal redefinición no es la redefinición de un método, sino un simple ejemplo de la sobrecarga de métodos.*



### Observación de ingeniería de software 27.6

*Cualquier objeto puede convertirse en una **String** con una llamada explícita o implícita al método **toString** del objeto.*



### Observación de ingeniería de software 27.7

*Toda clase debe redefinir el método **`toString`** para devolver información útil sobre los objetos de esa clase.*



### Error común de programación 27.5

*Si un método de una superclase y un método en su subclase tienen la misma firma pero diferente tipo de retorno, es un error de sintaxis.*

La aplicación (líneas 75 a 126) crea las instancias del objeto **`p`** de **`Punto`** y del objeto **`c`** de **`Circulo`** en las líneas 87 y 88 de **`main`**. Las representaciones **`String`** de cada objeto se adjuntan a **`String salida`** para mostrar que se inicializaron correctamente (líneas 90 y 91). Vea las dos primeras líneas de la salida de la captura de pantalla para confirmar esto.

La línea 95

```
refPunto = c;   // asigna Circulo a refPunto
```

asigna a **`Circulo c`** (una referencia hacia un objeto de subclase) a **`refPunto`** (una referencia de superclase). En Java, siempre es aceptable asignar una referencia de subclase a una referencia de superclase (debido a la relación de herencia *es un*). Un **`Circulo`** *es un* **`Punto`**, ya que la clase **`Circulo`** extiende a la clase **`Punto`**. Como veremos, asignar una referencia de superclase a una referencia de subclase es peligroso.

Las líneas 97 y 98 agregan el resultado de **`refPunto.toString()`** a la **`String salida`**. De manera interesante, cuando a esta **`refPunto`** se le envía al mensaje de **`toString`**, Java sabe que el objeto realmente es un **`Circulo`**, por lo que elige el método **`toString`** de **`Circulo`**, en lugar de usar el método **`toString`** de **`Punto`**, como pudo haber esperado. Éste es un ejemplo de *polimorfismo* y de *vinculación dinámica*, conceptos que trataremos con detalle más adelante en este capítulo. El compilador ve la expresión anterior y hace la pregunta "¿el tipo de dato de la referencia **`refPunto`** (es decir, **`Punto`**) tiene un método **`toString`** sin argumentos?" La respuesta a esta pregunta es sí (vea la definición de **`toString`** de **`Punto`** en la línea 35). El compilador simplemente verifica la sintaxis de la expresión y se asegura de que el método existe. En tiempo de ejecución, el intérprete hace la pregunta "¿de qué tipo es el objeto al que **`refPunto`** hace referencia?". Todo objeto en Java sabe su propio tipo de dato, por lo que la respuesta a esta pregunta es que **`refPunto`** hace referencia a un objeto **`Circulo`**. Basándose en esta respuesta, el intérprete llama al método **`toString`** del tipo de dato del objeto (es decir, el método **`toString`** de la clase **`Circulo`**). Vea la tercera línea de la salida para confirmar esto. Las dos principales técnicas que utilizamos para lograr este efecto son: 1) extender la clase **`Punto`** para crear la clase **`Circulo`**, y 2) redefinir el método **`toString`** con exactamente la misma firma en la clase **`Punto`** y en la clase **`Circulo`**.

La línea 102

```
refCirculo = (Circulo) refPunto;
```

convierte el tipo de **`refPunto`** (la cual admite hacer referencia a **`Circulo`** en este punto de la ejecución del programa) en un **`Circulo`**, y asigna el resultado a **`refCirculo`** (esta conversión de tipo sería peligrosa si **`refPunto`** realmente hiciera referencia a **`Punto`**, como explicaremos pronto). Después utilizamos **`refCirculo`** para agregar a **`String salida`** los diferentes hechos sobre la **`refCirculo`** de **`Circulo`**. Las líneas 104 y 105 invocan al método **`toString`** para agregar la representación **`String`** de **`Circulo`**. Las líneas 107 a 109 agregan el **`area`** del **`Circulo`** con el formato de una instancia de la clase **`DecimalFormat`** (paquete **`java.text`**) llamada **`precision2`** que da formato al número con dos dígitos a la derecha del punto decimal. El formato **`"0.00"`** (especificado en la línea 107) utiliza el **`0`** dos veces para indicar el número adecuado de dígitos después del punto decimal. Cada **`0`** es un lugar decimal requerido. El **`0`** a la izquierda del punto decimal indica un mínimo de un dígito a la izquierda del punto decimal.

Después, la estructura **`if/else`** de las líneas 113 a 118 intenta una conversión de tipo peligrosa en la línea 114. Convertimos el tipo de **`p`** de **`Punto`** en un **`Circulo`**. Si esto se intenta en tiempo de ejecución, Java determinaría que **`p`** realmente hace referencia a **`Punto`**, reconocería la conversión de tipo a **`Circulo`** como peligrosa, e indicaría una conversión inadecuada con el mensaje de **`ClassCastException`**. Sin embargo, evitamos que esta instrucción se ejecute con la condición **`if`**

```
if( p instanceof Circulo ) {
```

la cual utiliza el operador ***instanceof*** para determinar si el objeto al que **p** se refiere *es un* **Circulo**. Esta condición da como resultado **true** sólo si el objeto al que **p** se refiere *es un* **Circulo**; de lo contrario resulta en **false**. La referencia **p** no se refiere a un **Circulo**, por lo que la condición falla y se agrega una **String** a **salida**, la cual indica que **p** no se refiere a un **Circulo**.

Si eliminamos la prueba **if** del programa y lo ejecutamos, se genera el siguiente mensaje en tiempo de ejecución:

```
Exception in thread "main"
java.lang.ClassCastException: Punto
   at PruebaHerencia.main(PruebaHerencia.java:40)
```

Tales mensajes de error normalmente incluyen el nombre del archivo (**PruebaHerencia.java**) y el número de línea en la que ocurrió el error, para que pueda ir a esa línea específica del programa para depurarla. Observe que el número de línea especificado (**PruebaHerencia.java:40**) es diferente de los números de línea para el archivo **PruebaHerencia.java** que aparece en el texto. Esto se debe a que los ejemplos del texto están numerados consecutivamente para todos los archivos del mismo programa, con propósitos explicativos. Si abre el archivo **PruebaHerencia.java** en un editor, descubrirá que el error realmente ocurrió en la línea 40 (la cual es la línea 114 del programa completo).

## 27.5 Conversión implícita de un objeto de una subclase en un objeto de una superclase

A pesar del hecho de que un objeto de una subclase *es un* objeto de una superclase, el tipo de la subclase y el tipo de la superclase son diferentes. Los objetos de una subclase pueden tratarse como objetos de la superclase. Esto tiene sentido debido a que la subclase tiene miembros que corresponden a cada uno de los miembros de la superclase; recuerde que la subclase normalmente tiene más miembros que la superclase. La asignación en la otra dirección no está permitida, ya que asignar un objeto de una superclase a una referencia de una subclase dejaría indefinidos a los miembros adicionales de la subclase.

Una referencia a un objeto de una subclase se convertiría implícitamente en una referencia a un objeto de la superclase, ya que el objeto de la subclase *es un* objeto de la superclase a través de la herencia.

Existen cuatro posibles formas de mezclar y de hacer coincidir referencias de superclases y referencias de subclases con objetos de superclases y objetos de subclases:

1. Hacer referencia a un objeto de una superclase con una referencia de una superclase es directo.
2. Hacer referencia a un objeto de una subclase con una referencia de una subclase es directo.
3. Hacer referencia a un objeto de una subclase con una referencia de una superclase es seguro, ya que el objeto de una subclase también *es un* objeto de su superclase. Tal código sólo puede hacer referencia a miembros de la superclase. Si este código hace referencia sólo a miembros de la subclase, a través de referencias de superclase, el compilador reportará un error de sintaxis.
4. Hacer referencia a un objeto de una superclase con una referencia de subclase es un error de sintaxis. La referencia de subclase primero debe convertirse al tipo de una referencia de superclase.



**Error común de programación 27.6**

*Asignar un objeto de subclase a una referencia de superclase, y después intentar hacer referencia sólo a miembros de la subclase con la referencia de superclase, es un error de sintaxis.*

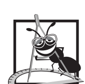
Aparentemente es conveniente tratar a los objetos de subclases como objetos de superclases, y hacer esto manipulando todos estos objetos con referencias de superclases aparentemente también es un problema. Por ejemplo, en un sistema de nómina nos gustaría recorrer un arreglo de empleados y calcular el pago semanal para cada persona. Sin embargo, la intuición nos dice que utilizar referencias de superclases permitiría al programa llamar únicamente a la rutina de superclase que calcula la nómina (si en realidad hay tal rutina en la superclase). Nosotros necesitamos una manera de invocar la rutina adecuada que calcule la nómina para cada objeto, ya sea un objeto de superclase o un objeto de subclase, y hacer esto simplemente utilizando la referencia de superclase. De hecho, es precisamente así como se comporta Java, y lo explicamos en este capítulo cuando consideramos el polimorfismo y la vinculación dinámica.

## 27.6 Ingeniería de software con herencia

Podemos utilizar la herencia para personalizar software existente. Cuando utilizamos la herencia para crear una nueva clase a partir de una clase existente, la nueva clase hereda los atributos y comportamientos de una clase existente, y después podemos agregar atributos y comportamientos o redefinir los comportamientos de una superclase para personalizar la clase con el objetivo de satisfacer nuestras necesidades.

Para los estudiantes puede resultar difícil apreciar los problemas que enfrentan los diseñadores y quienes implementan proyectos de software a gran escala para la industria. La gente experimentada en tales proyectos invariablemente afirma que una clave para mejorar el proceso de desarrollo de software es motivar la reutilización de software. La programación orientada a objetos en general, y Java en particular, ciertamente lo hacen.

Es la disponibilidad de bibliotecas substanciales y útiles la que proporciona los máximos beneficios de la reutilización de software a través de la herencia. Conforme se incremente el interés en Java, el interés en las bibliotecas de clases de Java se incrementará. Tal como el software producido por fabricantes independientes experimentó un gran crecimiento en la industria con la llegada de la computadora personal, así también sucederá con la creación y venta de las bibliotecas de clases de Java. Los diseñadores de aplicaciones construirán sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas se verán recompensados al tener incluidas sus bibliotecas en las aplicaciones. Lo que vemos venir es un compromiso masivo a nivel mundial para el desarrollo de bibliotecas de clases de Java, para una amplia variedad de aplicaciones.



**Observación de ingeniería de software 27.8**

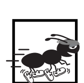
*Crear una subclase no afecta el código fuente de su superclase, o el código en bytes de las superclases de Java; la integridad de una superclase se preserva a través de la herencia.*

Una superclase especifica similitudes. Todas las clases derivadas de una superclase heredan las capacidades de esa superclase. En el proceso de diseño orientado a objetos, el diseñador busca similitudes entre un conjunto de clases y factores que necesita para formar superclases útiles. Las subclases entonces se personalizan más allá de las capacidades heredadas de las superclases.



**Observación de ingeniería de software 27.9**

*Así como el diseñador de sistemas no orientados a objetos deben evitar la proliferación de funciones innecesarias, el diseñador de sistemas orientados a objetos debe evitar la proliferación de clases innecesarias. La proliferación de clases genera problemas de administración y puede dificultar la reutilización de software, simplemente porque es más difícil para un usuario potencial de una clase localizar esa clase en una amplia colección. El equilibrio se encuentra en crear pocas clases que proporcionen funcionalidad adicional importante, sin embargo, dichas clases pueden ser demasiado ricas para ciertos usuarios.*



**Tip de rendimiento 27.1**

*Si las clases producidas a través de la herencia son más grandes de lo necesario, podrían desperdiciarse recursos de memoria y de procesamiento. Herede de la clase "que más se acerque" a lo que usted necesita.*

Observe que leer un conjunto de declaraciones de una subclase puede resultar confuso, ya que los miembros heredados no aparecen, pero dichos miembros están presentes en las subclases. Puede existir un problema similar en la documentación de las subclases.



**Observación de ingeniería de software 27.10**

*En un sistema orientado a objetos, con frecuencia las clases se encuentran muy relacionadas. "Ubique" los atributos y comportamientos comunes y colóquelos en una superclase. Después utilice la herencia para formar subclases para que no tenga que repetir atributos y comportamientos comunes.*



**Observación de ingeniería de software 27.11**

*Las modificaciones a una superclase no requieren que las subclases se modifiquen, mientras la interfaz pública de la superclase permanezca sin cambios.*

## 27.7 Composición *versus* herencia

Hemos explicado las relaciones *es un* que se implementan por herencia. También hemos explicado las relaciones *tiene un* (en ejemplos de capítulos anteriores) en la que una clase puede tener como miembros objetos de

otras clases; tales relaciones crean nuevas clases por medio de la *composición* de clases existentes. Por ejemplo, dadas las clases **`Empleado`**, **`FechaNacimiento`** y **`NumeroTelefonico`**, es inadecuado decir que un **`Empleado`** *es una* **`FechaNacimiento`** o que un **`Empleado`** *es un* **`NumeroTelefonico`**. Sin embargo, ciertamente es adecuado decir que un **`Empleado`** *tiene una* **`FechaNacimiento`** y que *tiene un* **`NumeroTelefonico`**.

## 27.8 Introducción al polimorfismo

Con el *polimorfismo*, es posible diseñar e implementar sistemas que sean más fácilmente *extensibles*. Los programas pueden escribirse para procesar genéricamente (como objetos de superclases) objetos de todas las clases existentes en una jerarquía. Las clases que no existen durante el desarrollo de un programa pueden agregarse con pocas o ninguna modificación a la parte genérica del programa; mientras esas clases sean parte de la jerarquía que se está procesando genéricamente. Las únicas partes de un programa que necesitan modificaciones son aquellas que requieran un conocimiento directo de una clase en particular que se agrega a la jerarquía. Estudiaremos dos jerarquías de clases importantes, y mostraremos cómo se manipulan de manera polimórfica los objetos a través de esas jerarquías.

## 27.9 Campos de tipo e instrucciones `switch`

Una manera de lidiar con objetos de diferentes tipos es por medio de una instrucción **`switch`** que realice la acción adecuada sobre cada objeto, basándose en el tipo de cada objeto. Por ejemplo, en una jerarquía de figuras en las que cada una tiene una variable de instancia **`tipoFigura`**, una estructura **`switch`** podría determinar a cuál método **`print`** llamar, basándose en el **`tipoFigura`** del objeto.

Existen muchos problemas con el uso de la lógica de **`switch`**. El programador podría olvidar hacer una prueba de tipos, cuando uno está garantizado. El programador podría olvidar probar todos los casos posibles de un **`switch`**. Si se modifica un sistema basado en **`switch`** agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en instrucciones **`switch`** existentes. Toda adición o eliminación de una clase demanda que cada instrucción **`switch`** en el sistema se modifique; rastrearlas puede llevarse demasiado tiempo y es propenso a errores.

Como veremos, la programación polimórfica puede eliminar la necesidad de la lógica de **`switch`**. El programador puede utilizar el mecanismo del polimorfismo de Java para realizar la lógica equivalente, con lo que eliminaría los tipos de errores generalmente asociados con la lógica de **`switch`**.



**Tip para prevenir errores 27.2**

*Una consecuencia interesante de utilizar el polimorfismo es que los programas adquieren una apariencia simplificada; contienen menos lógica de separación, a favor de un código secuencial más sencillo. Esta simplificación facilita el probar, depurar y mantener un programa.*

## 27.10 Método de vinculación dinámica

Suponga que un conjunto de clases de figuras como **`Circulo`**, **`Triangulo`**, **`Rectangulo`**, **`Cuadrado`**, etcétera, se derivan de la superclase **`Figura`**. En la programación orientada a objetos, cada una de estas clases puede dotarse con la habilidad de dibujarse a sí mismas. Cada clase tiene su propio método **`draw`**, y la implementación del método **`draw`** para cada figura es muy diferente. Cuando se dibuja una figura, cualquiera que ésta sea, sería bueno poder tratar a todas las figuras de manera genérica, como objetos de la superclase **`Figura`**. Después, para dibujar cualquier figura, podríamos simplemente llamar al método **`draw`** de la superclase **`Figura`**, y dejar al programa que determine dinámicamente (es decir, en tiempo de ejecución) cuál método **`draw`** de subclase utilizar, basándose en el tipo real del objeto.

Para permitir este tipo de comportamiento, declaramos **`draw`** en la superclase, y después redefinimos **`draw`** en cada una de las subclases para dibujar la figura adecuada.



**Observación de ingeniería de software 27.12**

*Cuando una subclase elige no redefinir un método, la subclase simplemente hereda la definición del método de su superclase inmediata.*

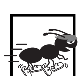
Si utilizamos una referencia de superclase para hacer referencia a un objeto de subclase e invocamos el método **`draw`**, el programa elegirá de manera dinámica (es decir, en tiempo de ejecución) el método **`draw`** de la subclase correcta. A esto se le llama *método de vinculación dinámica*, y lo ejemplificaremos en los ejemplos prácticos de este capítulo.

## 27.11 Métodos y clases `final`

Las variables pueden declararse como **`final`** para indicar que no pueden modificarse después de que se declaran, y que deben inicializarse cuando se declaran. También es posible definir métodos y clases con el modificador **`final`**.

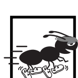
Un método que se declara **`final`** no puede redefinirse en una subclase. Los métodos que se declaran como **`static`** y los métodos que se declaran como **`private`**, son implícitamente **`final`**. La definición de un método **`final`** nunca puede cambiar, por lo que el compilador puede optimizar el programa eliminando las llamadas a métodos **`final`**, y reemplazarlas con el código ampliado con sus definiciones en cada ubicación de las llamadas al método; una técnica conocida como *poner en línea al código*.

Una clase que se declara como **`final`** no puede ser una superclase (es decir, una clase no puede heredar de una clase **`final`**). Todos los métodos de una clase **`final`** son implícitamente **`final`**.



**Tip de rendimiento 27.2**

*El compilador puede decidir poner en línea a una llamada a un método **`final`**, y lo hará para métodos **`final`** pequeños y sencillos. Colocarlas en línea no viola el encapsulamiento o el ocultamiento de información (pero mejora el rendimiento, ya que elimina la sobrecarga de realizar una llamada a un método).*



**Tip de rendimiento 27.3**

*Los preprocesadores canalizados pueden mejorar el rendimiento ejecutando simultáneamente diversas partes de las siguientes instrucciones, pero no si esas instrucciones siguen a una llamada a un método. Colocar en línea al código (lo que el compilador realiza en un método **`final`**) puede mejorar el rendimiento de estos preprocesadores, ya que elimina la transferencia de control fuera de línea asociada con una llamada a un método.*



**Observación de ingeniería de software 27.13**

*Una clase definida como **`final`** no puede extenderse, y cada uno de sus métodos es implícitamente **`final`**.*

## 27.12 Superclases abstractas y clases concretas

Cuando pensamos en una clase como un tipo, asumimos que los objetos de ese tipo serán instanciados. Sin embargo, existen casos en los que resulta útil definir clases cuyos objetos nunca intentará instanciar el programador. Dichas clases se conocen como *clases abstractas y contienen uno o más métodos abstractos*. Éstas se utilizan como superclases en situaciones de herencia, por lo que normalmente nos referimos a ellas como *superclases abstractas*. Ningún objeto de superclases abstractas pueden instanciarse.



**Error común de programación 27.7**

*Intentar crear una instancia de un objeto de una clase abstracta (es decir, una clase que contiene uno o más métodos abstractos), es un error de sintaxis.*



**Observación de ingeniería de software 27.14**

*Una clase abstracta puede tener datos de instancia y métodos no abstractos sujetos a las reglas normales de la herencia de las subclases. Una clase abstracta también pueden tener constructores.*

El único propósito de una clase abstracta es proporcionar una superclase apropiada de la que otras clases puedan heredar la interfaz y/o la implementación (en un momento veremos ejemplos de esto). Las clases cuyos objetos pueden instanciarse se conocen como *clases concretas*.



**Observación de ingeniería de software 27.15**

*Si una subclase se deriva de una superclase con un método **`abstract`**, y si no se proporciona una definición en la subclase para ese método **`abstract`** (es decir, si no se redefine ese método en la subclase), ese método permanece como **`abstract`** en la subclase. Como consecuencia, la subclase también es una clase **`abstract`**, y debe declararse explícitamente como **`abstract`**.*



**Observación de ingeniería de software 27.16**

*La habilidad de declarar un método* **`abstract`** *le da al diseñador de la clase suficiente poder sobre cómo implementará las subclases en una jerarquía de clases. Cualquier clase nueva que quiera heredar de esta clase es forzada a redefinir el método* **`abstract`** *(ya sea directamente o heredando de una clase que ha redefinido el método). De lo contrario, esa nueva clase contendrá un método* **`abstract`** *y, por lo tanto, será una clase* **`abstract`** *incapaz de instanciar objetos.*

Podríamos tener una superclase abstracta **`ObjetoBidimensional`** y derivar clases concretas como **`Cuadrado`**, **`Circulo`**, **`Triangulo`**, etcétera. También podríamos tener una superclase abstracta **`ObjetoTridimensional`** y derivar clases concretas como **`Cubo`**, **`Esfera`**, **`Cilindro`**, etcétera. Las superclases abstractas son demasiado genéricas para definir objetos reales; necesitamos ser más específicos antes de que podamos pensar en instanciar objetos. Por ejemplo, si alguien le pide que "dibuje la figura", ¿cuál dibujaría? Las clases concretas proporcionan las especificaciones que hacen razonable el crear instancias de objetos.

Se hace que una clase sea abstracta declarándola con la palabra reservada ***`abstract`***. Una jerarquía no necesita contener ninguna clase **`abstract`**, pero como veremos, muchos buenos sistemas orientados a objetos tienen jerarquías de clases encabezadas por superclases **`abstract`**. En algunos casos, las clases abstractas constituyen la cima de algunos niveles de la jerarquía. Un buen ejemplo de esto es la jerarquía de figuras de la figura 27.2. La jerarquía comienza con la superclase **`abstract Figura`**. En el siguiente nivel hacia abajo tenemos otras dos superclases abstractas, a saber, **`FiguraBidimensional`** y **`FiguraTridimensional`**. El siguiente nivel hacia abajo comenzaría definiendo las clases concretas para las figuras bidimensionales como **`Circulo`** y **`Cuadrado`**, y clases concretas para las figuras tridimensionales como **`Esfera`** y **`Cubo`**.

**Error común de programación 27.8**

*Si una clase con uno o más métodos* **`abstract`** *no se declara específicamente como* **`abstract`***, es un error de sintaxis.*

## 27.13 Ejemplo de polimorfismo

Aquí le presentamos un ejemplo de polimorfismo. Si una clase **`Rectangulo`** se deriva de la clase **`Cuadrilatero`**, entonces un objeto **`Rectangulo`** es una versión más específica del objeto **`Cuadrilatero`**. Una operación (como el cálculo del perímetro o el área) que puede realizarse sobre un objeto de la clase **`Cuadrilatero`** también puede realizarse sobre un objeto de la clase **`Rectangulo`**. Tales operaciones también pueden realizarse sobre otros "tipos de" **`Cuadrilateros`**, como **`Cuadrados`**, **`Paralelogramos`** y **`Trapezoides`**. Cuando se hace una solicitud para utilizar un método a través de una referencia de superclase, Java elige el método correcto redefinido de manera polimórfica en la subclase adecuada asociada con el objeto.

A través del polimorfismo, una llamada a un método puede provocar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada. Esto da al programador una tremenda capacidad de expresión. En las siguientes secciones veremos el poder del polimorfismo.



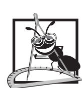
**Observación de ingeniería de software 27.17**

*Con el polimorfismo, el programador puede lidiar con las generalidades y deja que el ambiente en tiempo de ejecución se ocupe de lo específico. El programador puede ordenar que una amplia variedad de objetos se comporten de manera apropiada sin siquiera conocer los tipos de esos objetos.*



**Observación de ingeniería de software 27.18**

*El polimorfismo promueve la extensibilidad: El software escrito para invocar un comportamiento polimórfico se escribe de manera independiente a los tipos de los objetos a los que se envían los mensajes (es decir, llamadas a métodos). Por lo tanto, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en tales sistemas sin modificar el sistema base.*



**Observación de ingeniería de software 27.19**

*Si un método se declara como* **`final`***, éste no puede redefinirse en las subclases, por lo que las llamadas al método no pueden enviarse de manera polimórfica a los objetos de esas subclases. La llamada al método aún puede enviarse a las subclases, pero responderán de manera idéntica, en lugar de hacerlo de manera polimórfica.*



**Observación de ingeniería de software 27.20**

*Una clase* **`abstract`** *define una interfaz común para los diversos miembros de una jerarquía de clase. La clase* **`abstract`** *contiene métodos que se definirán en las subclases. Todas las clases de la jerarquía pueden utilizar esta misma interfaz a través del polimorfismo.*

Aunque no podemos crear instancias de objetos de superclases **`abstract`**, *podemos* declarar referencias a superclases **`abstract`**. Tales referencias pueden utilizarse para permitir manipulaciones polimórficas de objetos de subclases cuando tales objetos se instancian a partir de clases concretas.

Ahora consideremos más aplicaciones del polimorfismo. Un administrador de pantalla necesita desplegar una variedad de objetos, incluso nuevos tipos de objetos que se agregarán al sistema después de que esté escrito el administrador de pantalla. El sistema puede necesitar desplegar varias figuras (es decir, la superclase es **`Figura`**) como **`Circulo`**, **`Triangulo`**, **`Rectangulo`**, etcétera (cada clase de figura se deriva de la superclase **`Figura`**). El administrador de pantalla utiliza referencias de superclase (hacia **`Figura`**) para manipular los objetos a desplegar. Para dibujar cualquier objeto (independientemente del nivel en el que ese objeto aparezca en la jerarquía de herencia), el administrador de pantalla utiliza una referencia de superclase hacia el objeto, y simplemente envía un mensaje **`dibujar`** al objeto. El método **`dibujar`** se declaró como **`abstract`** en la superclase **`Figura`** y se redefinió en cada una de las subclases. Cada objeto de **`Figura`** sabe cómo dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse por el tipo de cada objeto, o si el administrador de pantalla ha visto antes objetos de ese tipo; el administrador simplemente le indica a cada objeto que se dibuje.

El polimorfismo es particularmente efectivo para implementar sistemas de software en capas. Por ejemplo, en sistemas operativos, cada tipo de dispositivo físico puede funcionar de manera muy diferente. Incluso, los comandos *leer* o *escribir* datos desde y hacia los dispositivos pueden tener cierta uniformidad. El mensaje *escribir* enviado a un objeto controlado por un dispositivo necesita interpretarse específicamente en el contexto de ese controlador de dispositivo, y en cómo es que ese controlador manipula los dispositivos de un tipo específico. Sin embargo, la llamada a *escribir* misma, en realidad no es diferente de *escribir* en cualquier otro dispositivo del sistema; simplemente coloca cierto número de bytes de la memoria en ese dispositivo. Un sistema operativo orientado a objetos podría utilizar una superclase **`abstract`** para proporcionar una interfaz adecuada para todos los controladores de dispositivos. Entonces, a través de la herencia de esa superclase **`abstract`**, se forman las subclases para que funcionen de manera similar. Las capacidades (es decir, la interfaz **`public`**) ofrecidas por los controladores de dispositivos se proporcionan como métodos **`abstract`** en la superclase **`abstract`**. Las implementaciones de estos métodos **`abstract`** se proporcionan en las subclases que corresponden a los tipos específicos de los controladores de dispositivos.

En la programación orientada a objetos es común definir una *clase iteradora* que pueda recorrer todos los objetos de un contenedor (como un arreglo). Por ejemplo, si desea imprimir una lista de objetos de una lista ligada, puede crearse una instancia de un objeto iterador que devuelva el siguiente elemento de la lista ligada cada vez que se llame al iterador. Los iteradores comúnmente se utilizan en la programación polimórfica para recorrer un arreglo o una lista ligada de objetos desde varios niveles de una jerarquía. Las referencias de dicha lista serían referencias de superclase. Una lista de objetos de una superclase de la clase **`FiguraBidimensional`** podría contener objetos de las clases **`Cuadrado`**, **`Circulo`**, **`Triangulo`**, etcétera. Enviar un mensaje **`dibujar`** a cada objeto de la lista podría, por medio del polimorfismo, dibujar la imagen correcta en la pantalla.

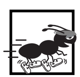
## 27.14 Nuevas clases y vinculación dinámica

El polimorfismo ciertamente funciona bien cuando todas las clases posibles se conocen por adelantado. Sin embargo, también funciona cuando se agregan nuevos tipos de clases a los sistemas.

Las nuevas clases se acomodan por medio del método de la vinculación dinámica (también llamada *vinculación tardía*). No es necesario conocer el tipo de un objeto en tiempo de compilación para que una llamada polimórfica se compile. En tiempo de ejecución, la llamada se hace coincidir con el método del objeto llamado.

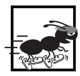
Un programa de administración de pantalla ahora puede manejar (sin tener que recompilar) nuevos tipos para desplegar objetos, conforme se agregan al sistema. La llamada al método **`dibujar`** permanece igual. Los nuevos objetos por sí mismos contienen un método **`dibujar`** que implementa las capacidades reales de dibujo.

Esto facilita el agregar nuevas capacidades al sistema con un impacto mínimo. También promueve la reutilización de software.



**Tip de rendimiento 27.4**

*Cuando el polimorfismo se implementa con el método de vinculación dinámica, es eficiente.*



**Tip de rendimiento 27.5**

*Los tipos de manipulaciones polimórficas que se hacen posibles con la vinculación dinámica, también pueden lograrse por medio de la lógica de* **`switch`** *codificada manualmente, de acuerdo con los campos de tipo de los objetos. El código polimórfico generado por el compilador de Java se ejecuta con un rendimiento comparable con la lógica de* **`switch`** *eficientemente codificada.*

## 27.15 Ejemplo práctico: Herencia de interfaz y de implementación

Ahora consideremos un ejemplo importante de herencia. Consideremos la jerarquía **`Punto`**, **`Circulo`**, **`Cilindro`** de la figura 27.4. Como cabeza de la jerarquía tenemos a la superclase **`abstract Figura`**. Esta jerarquía mecánicamente demuestra el poder del polimorfismo. En los ejercicios, exploramos una jerarquía de figuras más realista.

```
// Figura 27.4: Figura.java
// Definición de la clase base abstracta Figura

public abstract class Figura extends Object {
   public double area() { return 0.0; }
   public double volumen() { return 0.0; }
   public abstract String obtieneNombre();
}  // fin de la clase Figura
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Figura.java`**.

```
// Figura 27.4: Punto.java
// Definición de la clase Punto

public class Punto extends Figura {
   protected int x, y; // coordenadas del Punto

   // constructor sin argumentos
   public Punto() { establecePunto( 0, 0 ); }

   // constructor sin argumentos
   public Punto( int a, int b ) { establecePunto( a, b ); }

   // Establece las coordenada x y y de Punto
   public void establecePunto( int a, int b )
   {
      x = a;
      y = b;
   } // fin del método establecePunto

   // obtiene la coordenada x
   public int obtieneX() { return x; }

   // obtiene la coordenada x
   public int obtieneY() { return y; }

```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Punto.java`**. (Parte 1 de 2.)

```
    // convierte el punto a una representación String
    public String toString()
       { return “[“ + x + “, “ + y + “]”; }

    // devuelve el nombre de la clase
    public String obtieneNombre() { return “Punto”; }
} // fin de la clase Punto
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Punto.java`**. (Parte 2 de 2.)

```
// Figura 27.4: Circulo.java
// Definición de la clase Circulo

public class Circulo extends Punto {  // hereda de Punto
   protected double radio;

   // constructor sin argumentos
   public Circulo()
   {
      // llamada implícita al constructor de la superclase
      estableceRadio( 0 );
   } // fin del constructor Circulo

   // Constructor
   public Circulo( double r, int a, int b )
   {
      super( a, b );  // llama al constructor de la superclase
      estableceRadio( r );
   } // fin del constructor Circulo

   // Establece el radio del Circulo
   public void estableceRadio( double r )
      { radio = ( r >= 0 ? r : 0 ); }

   // Obtiene el radio del Circulo
   public double obtieneRadio() { return radio; }

   // Calcula el área del Circulo
   public double area() { return Math.PI * radio * radio; }

   // convierte Circulo a una String
   public String toString()
    { return “Centro = “ + super.toString() +
                “; Radio = “ + radio; }

   // devuelve el nombre de la clase
   public String obtieneNombre() { return “Circulo”; }
} // fin de la clase Circulo
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Circulo.java`**.

```
// Figura 27.4: Cilindro.java
// Definición de la clase Cilindro

public class Cilindro extends Circulo {
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Cilindro.java`**. (Parte 1 de 2.)

```
    protected double altura;  // altura del Cilindro

    // constructor sin argumentos
    public Cilindro()
    {
       // llamada implícita al constructor de la superclase
       estableceAltura( 0 );
} // fin del constructor Cilindro

    // constructor
    public Cilindro( double h, double r, int a, int b )
    {
       super( r, a, b );   // llama al constructor de la superclase
       estableceAltura( h );
    } // fin del constructor Cilindro

    // Establece la altura del Cilindro
    public void estableceAltura( double h )
       { altura = ( h >= 0 ? h : 0 ); }

    // Obtiene la altura del Cilindro
    public double obtieneAltura() { return altura; }

    // Calcula el área del Cilindro (es decir, la superficie)
    public double area()
    {
       return 2 * super.area() +
              2 * Math.PI * radio * altura;
    } // fin del método area

    // Calcula el volumen del Cilindro
    public double volumen() { return super.area() * altura; }

    // Convierte un Cilindro a una String
    public String toString()
       { return super.toString() + "; Altura = " + altura; }

    // Devuelve el nombre de la superclase
    public String obtieneNombre() { return "Cilindro"; }
} // fin de la clase Cilindro
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Cilindro.java`**. (Parte 2 de 2.)

```
// Figura 27.4: Prueba.java
// Controlador para la jerarquía Punto, Circulo, Cilindro
import javax.swing.JOptionPane;
import java.text.DecimalFormat;

public class Prueba {
    public static void main( String args[] )
    {
       Punto punto = new Punto( 7, 11 );
       Circulo circulo = new Circulo( 3.5, 22, 8 );
       Cilindro cilindro = new Cilindro( 10, 3.3, 10, 10 );
```

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Prueba.java`**. (Parte 1 de 2.)

```

      Figura arregloDeFiguras[];

      arregloDeFiguras = new Figura[ 3 ];

      // asigna arregloDeFiguras[0] al objeto de la subclase Punto
      arregloDeFiguras[ 0 ] = punto;

      // asigna arregloDeFiguras[1] al objeto de la subclase Circulo
      arregloDeFiguras[ 1 ] = circulo;

      // asigna arregloDeFiguras[2] al objeto de la subclase Cilindro
      arregloDeFiguras[ 2 ] = cilindro;

      String salida =
         punto.obtieneNombre() + “: “ + punto.toString() + “\n” +
         circulo.obtieneNombre() + “: “ + circulo.toString() + “\n” +
         cilindro.obtieneNombre() + “: “ + cilindro.toString();

      DecimalFormat precision2 = new DecimalFormat( “0.00” );

      // Realiza el ciclo a través de arregloDeFiguras e imprime el nombre,
      // el área, y el volumen de cada objeto.
      for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
         salida += “\n\n” +
            arregloDeFiguras[ i ].obtieneNombre() + “: “ +
            arregloDeFiguras[ i ].toString() +
            “\nArea = “ +
            precision2.format( arregloDeFiguras[ i ].area() ) +
            “\nVolumen = “ +
            precision2.format( arregloDeFiguras[ i ].volumen() );
      } // end for

      JOptionPane.showMessageDialog( null, salida,
         “Demostracion de polimorfismo”,
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // fin de main
} // fin de la clase Prueba
```

Demostracion de polimorfismo

Punto: [7, 11]
Circulo: Centro = [22, 8]; Radio = 3.5
Cilindro: Centro = [10, 10]; Radio = 3.3; Altura = 10.0

Punto: [7, 11]
Area = 0.00
Volumen = 0.00

Circulo: Centro = [22, 8]; Radio = 3.5
Area = 38.48
Volumen = 0.00

Cilindro: Centro = [10, 10]; Radio = 3.3; Altura = 10.0
Area = 275.77
Volumen = 342.12

Aceptar

**Figura 27.4** Jerarquía Figura, Punto, Circulo, Cilindro; **`Prueba.java`**. (Parte 2 de 2.)

**Figura** contiene el método **abstract obtieneNombre**, por lo que **Figura** debe declararse como una superclase **abstract**. **Figura** contiene otros dos métodos, **area** y **volumen**, cada uno de los cuales tiene una implementación que devuelve cero de manera predeterminada. **Punto** hereda estas implementaciones de **Figura**. Esto tiene sentido debido a que tanto el área como el volumen de un punto son cero. **Circulo** hereda el método volumen de **Punto**, pero **Circulo** proporciona su propia implementación del método **area**. **Cilindro** proporciona sus propias implementaciones de los métodos **area** (interpretada como la superficie del cilindro) y **volumen**.

En este ejemplo, la clase **Figura** se utiliza para definir un conjunto de métodos que todas las figuras de nuestra jerarquía tienen en común. Definir estos métodos en la clase **Figura** nos permite llamarlos de manera genérica a través de una referencia a **Figura**. Recuerde, los únicos métodos que pueden llamarse a través de cualquier referencia son los métodos públicos definidos en los tipos de clase declarados en la referencia y cualquier método público heredado en esa clase. Por lo tanto, podemos llamar a los métodos **Objeto** y **Figura**, a través de una referencia a **Figura**.

Observe que aunque **Figura** es una superclase **abstract**, aún contiene implementaciones de los métodos **area** y **volumen**, y estas implementaciones son heredables. La clase **Figura** proporciona una interfaz heredable (un conjunto de servicios) en la forma de tres métodos que todas las clases de la jerarquía contendrán. La clase **Figura** también proporciona algunas implementaciones que utilizarán las subclases de los primeros niveles de la jerarquía.

Este ejemplo práctico enfatiza que una subclase puede heredar la interfaz y/o la implementación de una superclase.



**Observación de ingeniería de software 27.21**

*Las jerarquías diseñadas para la herencia de la implementación tienden a tener a su funcionalidad arriba en la jerarquía; cada nueva subclase hereda uno o más de los métodos que se definieron en una superclase, y utiliza las definiciones de la superclase.*



**Observación de ingeniería de software 27.22**

*Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad más abajo en la jerarquía; una superclase especifica uno o más métodos que deben invocarse de manera idéntica para cada objeto en la jerarquía (es decir, tienen la misma firma), pero las subclases individuales proporcionan sus propias implementaciones de los métodos.*

La superclase **Figura** (figura 27.4, líneas 1 a 8) extiende a **Object**, consiste en tres métodos **public** y no contiene dato alguno (aunque podría). El método **obtieneNombre** es **abstract**, por lo que se redefine en las subclases. Los métodos **area** y **volumen** están definidos para que devuelvan **0.0**. Cuando es adecuado, estos métodos se redefinen en las subclases correspondientes a aquellas clases que tienen un cálculo de área diferente (clases **Circulo** y **Cilindro**) y/o un cálculo de volumen diferente (clase **Cilindro**).

La clase **Punto** (figura 27.4, líneas 9 a 40) se deriva de **Figura**. Un **Punto** tiene un área de 0.0 y un volumen de 0.0, por lo que los métodos **area** y **volumen** de la superclase no se redefinen aquí; ellos se heredan como se definió en **Figura**. Otros métodos incluyen **establecePunto** para asignar nuevas coordenadas **x** y **y** a un **Punto**, y **obtieneX** y **obtieneY** para devolver las coordenadas **x** y **y** de un **Punto**. El método **obtieneNombre** es una implementación del método **abstract** en la superclase. Si no se definiera este método, la clase **Punto** sería una clase **abstract**.

La clase **Circulo** (figura 27.4, líneas 41 a 78) se deriva de **Punto**. Un **Circulo** tiene un volumen de 0.0, por lo que el método de superclase **volumen** no se redefine; éste se hereda de la clase **Punto**, quien lo hereda de **Figura**. Un **Circulo** tiene un área diferente de un **Punto**, por lo que el método **area** se redefine. El método **obtieneNombre** es una implementación del método **abstract** de la superclase. Si este método no se redefine aquí, la versión de **obtieneNombre** de **Punto** se heredaría. Otros métodos incluyen **estableceRadio** para asignar un nuevo **radio** a un **Circulo**, y **obtieneRadio** para devolver el **radio** de un **Circulo**.



**Observación de ingeniería de software 27.23**

*Una subclase siempre hereda la versión definida más recientemente de cada método **public** y **protected** de sus superclases directa e indirecta.*

La clase **Cilindro** (figura 27.4, líneas 79 a 122) se deriva de **Circulo**. Un **Cilindro** tiene un área y un volumen diferente de aquellos de la clase **Circulo**, por lo que los métodos **area** y **volumen** se redefinen. El método **obtieneNombre** es una implementación del método **abstract** de la superclase. Si este método no se ha redefinido aquí, se hereda la versión de **obtieneNombre** de **Circulo**. Otros métodos incluyen **estableceAltura** para asignar una nueva **altura** a un **Cilindro**, y **obtieneAltura** para devolver la **altura** de un **Cilindro**.

El método **main** de la clase **Prueba** (figura 27.4, líneas 123 a 173) crea la instancia del objeto **punto** de **Punto**, del objeto **circulo** de **Circulo** y del objeto **cilindro** de **Cilindro** (líneas 131 a 133). Después, se instancia el arreglo **arregloDeFiguras** (línea 137). Este arreglo de referencias de la superclase **Figura** se referirá a cada objeto **instanciado** de la subclase. En la línea 140, la referencia **punto** se asigna al elemento **arregloDeFiguras[0]** del arreglo. En la línea 143, la referencia **circulo** se asigna al elemento **arregloDeFiguras[1]** del arreglo. En la línea 146, la referencia **cilindro** se asigna al elemento **arregloDeFiguras[2]** del arreglo. Ahora, cada referencia de la superclase **Figura** en el arreglo se refiere a un objeto de la subclase del tipo **Punto**, **Circulo** o **Cilindro**.

Las líneas 148 a 151 invocan a los métodos **obtieneNombre** y **toString** para ilustrar que los objetos se inicializan correctamente (vea las tres primeras líneas de la salida).

Después, la estructura **for** de las líneas 157 a 165 recorre el **arregloDeFiguras** y se hacen las siguientes llamadas, durante cada iteración del ciclo:

```
arregloDeFiguras[ i ].obtieneNombre()
arregloDeFiguras[ i ].toString()
arregloDeFiguras[ i ].area()
arregloDeFiguras[ i ].volumen()
```

Cada una de estas llamadas a métodos se invoca en el objeto al que **arregloDeFiguras** actualmente hace referencia. Cuando el compilador ve cada una de estas llamadas, simplemente intenta determinar si una referencia a **Figura** (**arregloDeFiguras[ i ]**) puede utilizarse para llamar a estos métodos. Para los métodos **obtieneNombre**, **area** y **volumen**, la respuesta es sí, ya que cada uno de estos métodos está definido en la clase **Figura**. Para el método **toString**, el compilador primero ve la clase **Figura** y determina que **toString** no está definido ahí, después el compilador continúa con la superclase **Figura** (**Object**) para determinar si **Figura** hereda un método **toString** que no tome argumentos (lo cual es cierto, ya que todos los **Objects** tienen un método **toString**).

La salida ilustra que los cuatro métodos se invocan adecuadamente, basándose en el tipo del objeto al que se hizo referencia. Primero, se despliega la cadena **"Punto: "** y las coordenadas del objeto **punto** (**arregloDeFiguras[0]**); el área y el volumen son **0**. Después, se despliega la cadena **"Circulo: "**, las coordenadas del objeto **circulo**, y el **radio** del objeto **circulo** (**arregloDeFiguras[1]**); el área del **circulo** se calcula, y el volumen es **0**. Por último, se despliega la cadena **"Cilindro: "**, las coordenadas del objeto **cilindro**, el **radio** del objeto **cilindro** y la **altura** del objeto **cilindro** (**arregloDeFiguras[2]**); el área y el volumen del **cilindro** se calculan. Todas las llamadas a los métodos **obtieneNombre**, **toString**, **area** y **volumen** se resuelven en tiempo de ejecución con vinculación dinámica.

## 27.16 Ejemplo práctico: Creación y uso de interfaces

Nuestro siguiente ejemplo (figura 27.5) reexamina la jerarquía **Punto**, **Circulo**, **Cilindro**, y reemplaza a la superclase **abstract Figura** con la interfaz **Figura**. Una definición de interfaz comienza con la palabra reservada ***interface*** y contiene un conjunto de métodos **public** y **abstract**. Las interfaces también pueden contener datos **public final static**. Para utilizar una interfaz, una clase debe especificar que la implementa y debe definir cada método en la interfaz con el número de argumentos y el tipo de retorno especificado en la definición de la interfaz. Si la clase deja indefinido un método en la interfaz, la clase se vuelve **abstract** y debe declararse como tal en la primera línea de la definición de su clase. Implementar una interfaz es como firmar un contrato con el compilador, el cual establece que "definiré todos los métodos especificados por la interfaz".



**Error común de programación 27.9**

*Dejar indefinido un método de una interfaz, en una clase que implementa la interfaz, da como resultado un error de compilación que indica que la clase debe declararse como* ***abstract****.*

Por lo general, una interfaz se utiliza en lugar de una clase abstracta, cuando no hay una implementación predeterminada a heredar; es decir, no hay variables de instancia ni implementaciones predeterminadas de métodos. Como las clases **`public abstract`**, las interfaces en general son tipos de datos **`public`**, por lo que se definen a sí mismos en archivos con el mismo nombre que la interfaz y la extensión **`.java`**.

La definición de la interfaz **`Figura`** comienza en la línea 4. La interfaz **`Figura`** tiene los métodos **`abstract area`**, **`volumen`** y **`obtieneNombre`**. Como coincidencia, los tres métodos no toman argumentos. Sin embargo, éste no es un requerimiento de los métodos en una interfaz.

```
1  // Figura 27.5: Figura.java
2  // Definición de la interfaz Figura
3
4  public interface Figura {
5     public abstract double area();
6     public abstract double volumen();
7     public abstract String obtieneNombre();
8  } // fin de la interfaz Figura
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura; Figura.java`**.

```
9  // Figura 27.5: Punto.java
10 // Definición de la clase Punto
11
12 public class Punto extends Object implements Figura {
13    protected int x, y; // coordenadas del Punto
14
15    // constructor sin argumentos
16    public Punto() { establecePunto( 0, 0 ); }
17
18    // constructor
19    public Punto( int a, int b ) { establecePunto( a, b ); }
20
21    // Establece las coordenadas x y y de Punto
22    public void establecePunto( int a, int b )
23    {
24       x = a;
25       y = b;
26    } // fin del método establecePunto
27
28    // obtiene la coordena x
29    public int obtieneX() { return x; }
30
31    // obtiene la coordena y
32    public int obtieneY() { return y; }
33
34    // convierte el punto a una representación a String
35    public String toString()
36       { return “[“ + x + “, “ + y + “]”; }
37
38    // devuelve el área
39    public double area() { return 0.0; }
40
41    // devuelve el volumen
42    public double volumen() { return 0.0; }
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Punto.java`**. (Parte 1 de 2.)

```
43
44      // devuelve el nombre de la clase
45      public String obtieneNombre() { return "Punto"; }
46   } // fin de la clase Punto
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Punto.java`**. (Parte 2 de 2.)

La línea 12

```
public class Punto extends Object implements Figura {
```

indica que la clase **`Punto`** extiende a la clase **`Object`** e implementa la interfaz **`Figura`**. La clase **`Punto`** proporciona definiciones de los tres métodos en la interfaz. El método **`area`** está definido en la línea 39, el método **`volumen`** está definido en la línea 42, y el método **`obtieneNombre`** está definido en la línea 45. Estos tres métodos satisfacen el requerimiento de la implementación para los tres métodos definidos en la interfaz. Hemos cumplido con el contrato con el compilador.

Cuando una clase implementa una interfaz, aplica la misma relación *es un* provista por la herencia. En nuestro ejemplo, la clase **`Punto`** implementa a **`Figura`**. Por lo tanto, un objeto **`Punto`** *es una* **`Figura`**. De hecho, los objetos de cualquier clase que extienden a **`Punto`**, también son objetos de **`Figura`**. A través de esta relación, hemos mantenido las definiciones originales de la clase **`Circulo`**, de la clase **`Cilindro`**, y de la clase de aplicación **`Prueba`** de la figura 27.4, para mostrar que se puede utilizar una interfaz en lugar de una clase **`abstract`**, para procesar de manera polimórfica unas **`Figuras`**. Observe que la salida del programa es idéntica a la de la figura 27.4. También observe que el método **`toString`** de **`Object`** es invocado a través de una referencia a la interfaz **`Figura`** (línea 166).

```
// Figura 27.5: Circulo.java
// Definición de la clase Circulo

public class Circulo extends Punto {  // hereda desde Punto
   protected double radio;

   // constructor sin argumentos
   public Circulo()
   {
      // llamada implícita al constructor de la superclase
      estableceRadio( 0 );
   } // fin del constructor Circulo

   // Constructor
   public Circulo( double r, int a, int b )
   {
      super( a, b );  // llamada al constructor de la superclase
      estableceRadio( r );
   } // fin del constructor Circulo

   // Establece el radio del Circulo
   public void estableceRadio( double r )
      { radio = ( r >= 0 ? r : 0 ); }

   // Obtiene el radio del Circulo
   public double obtieneRadio() { return radio; }

   // Calcula el área del Círculo
   public double area() { return Math.PI * radio * radio; }
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Circulo.java`**. (Parte 1 de 2.)

```

   // convierte el Circulo a una String
   public String toString()
      { return “Centro = “ + super.toString() +
               “; Radio = “ + radio; }

   // devuelve el nombre de la clase
   public String obtieneNombre() { return “Circulo”; }
} // fin de la clase Circulo
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Circulo.java`**. (Parte 2 de 2.)

```
// Figura 27.5: Cilindro.java
// Definición de la clase Cilindro

public class Cilindro extends Circulo {
   protected double altura;  // altura del Cilindro

   // constructor sin argumentos
   public Cilindro()
   {
      // llamada implícita al constructor de la superclase
      estableceAltura( 0 );
   } // fin del constructor Cilindro

   // constructor
   public Cilindro( double h, double r, int a, int b )
   {
      super( r, a, b );   // llama al constructor de la superclase
      estableceAltura( h );
   } // fin del constructor Cilindro

   // Establece la altura del Cilindro
   public void estableceAltura( double h )
      { altura = ( h >= 0 ? h : 0 ); }

   // Obtiene la altura del Cilindro
   public double obtieneAltura() { return altura; }

   // Calcula el área del Cilindro (es decir, el área de la superficie)
   public double area()
   {
      return 2 * super.area() +
               2 * Math.PI * radio * altura;
   } // fin del método area

   // Calcula el volumen del Cilindro
   public double volumen() { return super.area() * altura; }

   // Convierte un Cilindro a una String
   public String toString()
      { return super.toString() + “; Altura = “ + altura; }

```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Cilindro.java`**. (Parte 1 de 2.)

```
    // Devuelve el nombre de la clase
    public String obtieneNombre() { return “Cilindro”; }
} // fin de la clase Cilindro
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz de **`Figura`**; **`Cilindro.java`**. (Parte 2 de 2.)



### Observación de ingeniería de software 27.24

*Todos los métodos de la clase* **`Object`** *pueden invocarse por medio de una referencia a un tipo de dato interfaz; una referencia se refiere a un objeto, y todos los objetos tienen los métodos definidos por la clase* **`Object`**.

Un beneficio de utilizar interfaces es que una clase puede implementar tantas interfaces como sea necesario, además de extender una clase. Para implementar más de una interfaz, simplemente proporcione una lista separada por comas con los nombres de las interfaces, después de la palabra reservada **`implements`** en la definición de la clase. Esto es particularmente útil en el mecanismo de manipulación de eventos GUI. Una clase que implementa más de una interfaz que escucha eventos (como la **`ActionListener`** de los ejemplos anteriores) puede procesar diferentes tipos de eventos GUI, como veremos en el capítulo 29.

```
// Figura 27.5: Prueba.java
// Controlador para la jerarquía Punto, Circulo, Cilindro
import javax.swing.JOptionPane;
import java.text.DecimalFormat;

public class Prueba {
   public static void main( String args[] )
   {
      Punto punto = new Punto( 7, 11 );
      Circulo circulo = new Circulo( 3.5, 22, 8 );
      Cilindro cilindro = new Cilindro( 10, 3.3, 10, 10 );

      Figura arregloDeFiguras[];

      arregloDeFiguras = new Figura[ 3 ];

      // asigna arregloDeFiguras[0] al objeto de la subclase Punto
      arregloDeFiguras[ 0 ] = punto;

      // asigna arregloDeFiguras[0] al objeto de la subclase Circulo
      arregloDeFiguras[ 1 ] = circulo;

      // asigna arregloDeFiguras[0] al objeto de la subclase Cilindro
      arregloDeFiguras[ 2 ] = cilindro;

      String salida =
         punto.obtieneNombre() + “: “ + punto.toString() + “\n” +
         circulo.obtieneNombre() + “: “ + circulo.toString() + “\n” +
         cilindro.obtieneNombre() + “: “ + cilindro.toString();

      DecimalFormat precision2 = new DecimalFormat( “#0.00” );

      // Ciclo a través de arregloDeFiguras e impresión del nombre,
      // el área, y el volumen de cada objeto.
      for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
         salida += “\n\n” +
```

**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz **`Figura`**; **`Prueba.java`**. (Parte 1 de 2.)

```
               arregloDeFiguras[ i ].obtieneNombre() + ": " +
               arregloDeFiguras[ i ].toString() +
               "\nArea = " +
               precision2.format( arregloDeFiguras[ i ].area() ) +
               "\nVolumen = " +
               precision2.format( arregloDeFiguras[ i ].volumen() );
         }

         JOptionPane.showMessageDialog( null, salida,
            "Demostracion de polimorfismo",
            JOptionPane.INFORMATION_MESSAGE );

         System.exit( 0 );
      } // fin de main
   } // fin de la clase Prueba
```

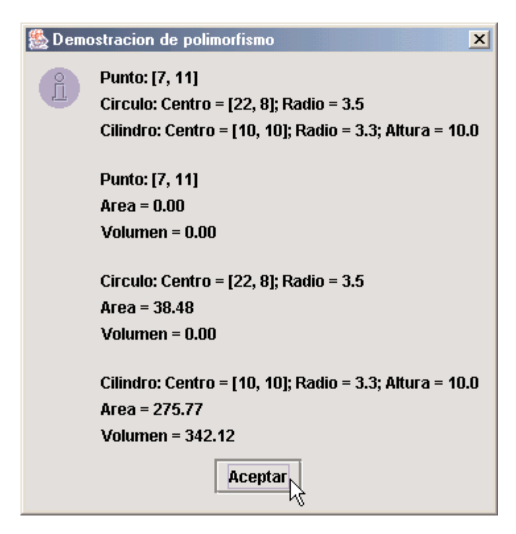



**Figura 27.5** Jerarquía Punto, Circulo, Cilindro con una interfaz **`Figura`**; **`Prueba.java`**. (Parte 2 de 2.)

Otro uso de las interfaces es definir un conjunto de constantes que pueden utilizarse en muchas definiciones de clases. Considere la interfaz **`Constantes`**

```
public interface Constantes {
   public static final int UNO = 1;
   public static final int DOS = 2;
   public static final int TRES = 3;
}
```

Las clases que implementan la interfaz **`Constantes`** pueden utilizar las constantes **`UNO`**, **`DOS`** y **`TRES`** en cualquier parte de la definición de la clase. Una clase puede incluso utilizar estas constantes, simplemente importando la interfaz, y después refiriéndose a cada constante como **`Constantes.UNO`**, **`Constantes.DOS`** y **`Constantes.TRES`**. Ningún método está declarado en esta interfaz, por lo que a una clase que implementa la interfaz no se le solicita que proporcione implementación alguna.

## 27.17 Definiciones de clases internas

Todas las definiciones de clases que hemos explicado hasta este punto, se definieron con alcance de archivo; las clases se definieron en archivos, pero no dentro de otras clases de esos archivos. Java proporciona una facilidad llamada *clases internas*, en las que las clases pueden definirse dentro de otras clases. Tales clases pueden ser definiciones completas de clases, o definiciones de *clases internas anónimas* (clases sin un nombre). Las clases

internas se utilizan principalmente en la manipulación de eventos. Sin embargo, tienen otros beneficios. Por ejemplo, cuando se define un tipo de dato abstracto cola, se puede utilizar una clase interna para representar los objetos que almacena cada elemento actualmente en la cola. Sólo la estructura de datos cola requiere saber cómo se almacenan los objetos de manera interna, por lo que la implementación puede ocultarse definiendo una clase interna como parte de la clase **Cola**.

Las clases internas con frecuencia se utilizan con la manipulación de eventos GUI, por lo que aprovechamos esta oportunidad, no sólo para mostrarle las definiciones de clases internas, sino para también demostrarle una aplicación que se ejecuta en su propia ventana. Una vez que complete este ejemplo, podrá utilizar en sus aplicaciones las técnicas GUI que hasta el momento hemos mostrado sólo en applets.

Para demostrar una definición de una clase interna, la figura 27.6 utiliza una versión simplificada de la clase **Hora2** (renombrada aquí como **Hora**) correspondiente a la figura 26.3. La clase **Hora** proporciona un constructor predeterminado, los mismos métodos *establecer*/*obtener* de la figura 26.3, y un método **toString**. Además, este programa define la clase **VentanaPruebaHora** como una aplicación. La aplicación se ejecuta en su propia ventana.

```
// Figura 27.6: Hora.java
// Definición de la clase Hora
import java.text.DecimalFormat;  // se utiliza para dar formato a números

// Esta clase contiene la hora en formato de 24 horas
public class Hora extends Object {
   private int hora;     // 0 - 23
   private int minuto;   // 0 - 59
   private int segundo;  // 0 - 59

   // el constructor Hora inicializa cada variable de instancia
   // en cero. Asegura que el objeto Hora se encuentra en un
   // estado consistente
   public Hora() { estableceHora( 0, 0, 0 ); }

   // Establece un nuevo valor de hora con el formato universal. Realiza
   // las validaciones de los datos. Establece en cero los valores no válidos.
   public void estableceHora( int h, int m, int s )
   {
      estableceHora( h );     // establece la hora
      estableceMinuto( m );   // establece el minuto
      estableceSegundo( s );  // establece el segundo
   } // fin del método estableceHora

   // establece la hora
   public void estableceHora( int h )
      { hora = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

   // establece el minuto
   public void estableceMinuto( int m )
      { minuto = ( ( m >= 0 && m < 60 ) ? m : 0 ); }

   // establece el segundo
   public void estableceSegundo( int s )
      { segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

   // obtiene la hora
   public int obtieneHora() { return hora; }
```

**Figura 27.6** Demostración de una clase interna en una aplicación de visualización; **Hora.java**. (Parte 1 de 2.)

```

   // obtiene el minuto
   public int obtieneMinuto() { return minuto; }

   // obtiene el segundo
   public int obtieneSegundo() { return segundo; }

   // Conversión a una String en formato de hora estándar
   public String toString()
   {
      DecimalFormat dosDigitos = new DecimalFormat( “00” );

      return ( ( obtieneHora() == 12 || obtieneHora() == 0 ) ?
                 12 : obtieneHora() % 12 ) + “:” +
               dosDigitos.format( obtieneMinuto() ) + “:” +
               dosDigitos.format( obtieneSegundo() ) +
               ( obtieneHora() < 12 ? “ AM” : “ PM” );
   } // fin del método toString
} // fin de la clase Hora
```

**Figura 27.6** Demostración de una clase interna en una aplicación de visualización; **`Hora.java`**. (Parte 2 de 2.)

```
// Figura 27.6: VentanaPruebaHora.java
// Demostración de los métodos establecer y obtener de la clase Hora
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class VentanaPruebaHora extends JFrame {
   private Hora h;
   private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
   private JTextField campoHora, campoMinuto,
                      campoSegundo, desplegar;
   private JButton botonSalida;

   public VentanaPruebaHora()
   {
      super( “Demostración de la clase Interna” );

      h = new Hora();

      Container c = getContentPane();

      // crea una instancia de la clase interna
      ActionEventHandler manipulador = new ActionEventHandler();

      c.setLayout( new FlowLayout() );
      etiquetaHora = new JLabel( “Establece la hora” );
      campoHora = new JTextField( 10 );
      campoHora.addActionListener( manipulador );
      c.add( etiquetaHora );
      c.add( campoHora );
```

**Figura 27.6** Demostración de una clase interna en una aplicación de visualización; **`VentanaPruebaHora.java`**. (Parte 1 de 3.)

```

      etiquetaMinuto = new JLabel( “Establece el minuto” );
      campoMinuto = new JTextField( 10 );
      campoMinuto.addActionListener( manipulador );
      c.add( etiquetaMinuto );
      c.add( campoMinuto );

      etiquetaSegundo = new JLabel( “Establece el segundo” );
      campoSegundo = new JTextField( 10 );
      campoSegundo.addActionListener( manipulador );
      c.add( etiquetaSegundo );
      c.add( campoSegundo );

      desplegar = new JTextField( 30 );
      desplegar.setEditable( false );
      c.add( desplegar );

      botonSalida = new JButton( “Salir” );
      botonSalida.addActionListener( manipulador );
      c.add( botonSalida );
   } // fin del constructor VentanaPruebaHora

   public void despliegaHora()
   {
      desplegar.setText( “La hora es: “ + h );
   } // fin del método despliegaHora

   public static void main( String args[] )
   {
      VentanaPruebaHora ventana = new VentanaPruebaHora();

      ventana.setSize( 400, 140 );
      ventana.show();
   } // fin de main

   // Definición de la clase interna para la manipulación de eventos
   private class ActionEventHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
           if ( e.getSource() == botonSalida )
            System.exit( 0 );   // termina la aplicación
         else if ( e.getSource() == campoHora ) {
            h.estableceHora(
               Integer.parseInt( e.getActionCommand() ) );
            campoHora.setText( “” );
         }
         else if ( e.getSource() == campoMinuto ) {
            h.estableceMinuto(
               Integer.parseInt( e.getActionCommand() ) );
            campoMinuto.setText( “” );
         }
         else if ( e.getSource() == campoSegundo ) {
            h.estableceSegundo(
               Integer.parseInt( e.getActionCommand() ) );
```

**Figura 27.6** Demostración de una clase interna en una aplicación de visualización; **`VentanaPruebaHora.java`**. (Parte 2 de 3.)

```
142                campoSegundo.setText( "" );
143             }
144 
145             despliegaHora();
146          } // fin del método accionRealizada
147       } // fin de la clase ManipDeEventos
148    } // fin de la clase VentanaPruebaHora
```

Cuadro Cerrar

Cuadro Maximizar

Cuadro Minimizar

Barra de Título

**Figura 27.6** Demostración de una clase interna en una aplicación de visualización; **`VentanaPruebaHora.java`**. (Parte 3 de 3.)

La línea 64

```
public class VentanaPruebaHora extends JFrame {
```

indica que la clase **`VentanaPruebaHora`** extiende a la clase ***`JFrame`*** (del paquete **`javax.swing`**), en lugar de la clase **`JApplet`** (como muestra la figura 26.3). La superclase **`JFrame`** proporciona los atributos y comportamientos básicos de una ventana; una *barra de título* y botones para *minimizar*, *maximizar* y *cerrar* la

ventana (todos etiquetados en la primera captura de pantalla). La clase **VentanaPruebaHora** utiliza los mismos componentes GUI que el applet de la figura 26.3, con la excepción de que al botón (línea 69) ahora se le llama **botonSalida**, y se utiliza para finalizar la aplicación.

El método **init** del applet se reemplazó con un constructor (línea 71) para garantizar que los componentes GUI de la ventana se crean conforme comienza la ejecución. El método **main** (línea 115) define un objeto **new** de la clase **VentanaPruebaHora** que da como resultado una llamada al constructor. Recuerde, **init** es un método especial, cuya invocación está garantizada cuando un applet comienza su ejecución. Sin embargo, este programa no es un applet, por lo que no se garantiza que el método **init** sea llamado.

En el constructor aparecen diversas características nuevas. La línea 73 llama al constructor de la superclase **JFrame** con la cadena "**Demostracion de una clase interna**". Esta cadena se despliega en la barra de título de la ventana por medio del constructor de **JFrame**. La línea 80

```
ActionEventHandler manipulador = new ActionEventHandler();
```

define dos instancias de nuestra clase **ActionEventHandler** y la asigna a **manipulador**. Esta referencia se pasa a cada uno de las cuatro llamadas a **ActionListener** (líneas 85, 91, 97 y 106) que registran los manipuladores de eventos para cada componente GUI que genera los eventos en el ejemplo (**campoHora**, **campoMinuto**, **campoSegundo**, y **botonSalida**). Cada llamada a **addActionListener** requiere un objeto de **ActionListener** para pasarlo como argumento. En realidad, **manipulador** es un **ActionListener**. La línea 124 (la primera línea de la definición de la clase interna)

```
private class ActionEventHandler implements ActionListener {
```

indica que la clase interna **ActionEventHandler** implementa a **ActionListener**. Así, cada objeto de tipo **ActionEventHandler** *es un* **ActionListener**. ¡Se satisface el requerimiento que indica que **addActionListener** se pase como un objeto de tipo **ActionListener**! Ésta *es una* relación que se utiliza de manera extensiva en el mecanismo de manipulación de eventos del GUI, como lo verá en los siguientes capítulos. La clase interna se define como **private** debido a que solamente se utilizará en la definición de la clase. Las clases internas pueden ser **private**, **protected** o **public**.

Un objeto de la clase interna tiene una relación especial con el objeto de la clase externa que lo crea. Al objeto de la clase interna se le permite tener acceso directo a todas las variables de instancia y a los métodos del objeto de la clase externa. El método **actionPerformed** (línea 125) de la clase **EventHandler** hace justamente eso. A través del método, se utilizan las variables de instancia **h**, **botonSalida**, **campoHora**, **campoMinuto**, así como su método **despliegaHora**. Observe que ninguno de éstos es un "manipulador" del objeto de la clase externa. Ésta es una relación libre creada por el compilador entre la clase externa y sus clases internas.



### Observación de ingeniería de software 27.25

*A un objeto de la clase interna se le permite tener acceso directo a las variables de instancia y a los métodos del objeto de la clase externa que la define.*

[*Nota:* Esta aplicación se debe finalizar al presionar el botón **Entrar**. Recuerde, una aplicación que despliega una ventana debe terminar con una llamada a **System.exit( 0 )**. Observe además que una ventana en Java tiene 0 pixeles de ancho y 0 pixeles de alto y no se despliega de manera predeterminada, Las líneas 119 y 120 redimensionan el tamaño de la ventana y la muestran en la pantalla.]

Una clase interna también puede definirse dentro del método de una clase. Tal clase interna tiene acceso a los miembros de la clase externa. Sin embargo, tiene acceso limitado a las variables locales del método en el cual está definida.

### Observación de ingeniería de software 27.26

*Una clase interna definida dentro de un método tiene acceso directo a todas las variables de instancia y métodos del objeto de la clase externa en la que se define y en cualquier variable local de* **final** *en el método.*

La aplicación de la figura 27.7 modifica la clase **VentanaPruebaHora** para utilizar las *clases anónimas internas* definidas dentro de métodos. Una clase anónima interna no tiene nombre, de modo que se debe crear la clase anónima interna en el punto en donde se define la clase dentro del programa. En este ejemplo, demostramos las clases anónimas internas de dos formas. Primero, separamos las clases anónimas internas que

implementan una interfaz (**ActionListener**) para crear manipuladores para cada uno de los tres **JTextFields** **campoHora**, **campoMinuto** y **campoSegundo**. También demostramos cómo terminar una aplicación cuando el usuario hace clic en el cuadro **Cerrar** en la ventana. El manipulador de eventos se define como la clase anónima interna que extiende a la clase (**WindowAdapter**). La clase **Hora** es idéntica a la figura 27.6, de modo que no se incluye aquí. Además, el botón **Salir** se eliminó de este ejemplo.

```
// Figura 27.7: VentanaPruebaHora.java
// Demostración de los métodos establecer y obtener para la clase Hora
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class VentanaPruebaHora extends JFrame {
   private Hora h;
   private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
   private JTextField campoHora, campoMinuto,
                      campoSegundo, despliega;

   public VentanaPruebaHora()
   {
      super( "Demostración de la clase interna" );

      h = new Hora();

      Container c = getContentPane();

      c.setLayout( new FlowLayout() );
      etiquetaHora = new JLabel( "Establece la hora" );
      campoHora = new JTextField( 10 );
      campoHora.addActionListener(
         new ActionListener() {  // clase interna anónima
            public void actionPerformed( ActionEvent e )
            {
               h.estableceHora(
                  Integer.parseInt( e.getActionCommand() ) );
               campoHora.setText( "" );
               despliegaHora();
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( etiquetaHora );
      c.add( campoHora );

      etiquetaMinuto = new JLabel( "Establece el minuto" );
      campoMinuto = new JTextField( 10 );
      campoMinuto.addActionListener(
         new ActionListener() {  // clase interna anónima
            public void actionPerformed( ActionEvent e )
            {
               h.estableceMinuto(
                  Integer.parseInt( e.getActionCommand() ) );
               campoMinuto.setText( "" );
```

**Figura 27.7** Demostración de las clases anónimas internas; **VentanaPruebaHora.java**. (Parte 1 de 3).

```
               despliegaHora();
            }
         }
      );
      c.add( etiquetaMinuto );
      c.add( campoMinuto );

      etiquetaSegundo = new JLabel( "Establece el segundo" );
      campoSegundo = new JTextField( 10 );
      campoSegundo.addActionListener(
         new ActionListener() {  // clase interna anónima
            public void actionPerformed( ActionEvent e )
            {
               h.estableceSegundo(
                  Integer.parseInt( e.getActionCommand() ) );
               campoSegundo.setText( "" );
               despliegaHora();
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( etiquetaSegundo );
      c.add( campoSegundo );

      despliega = new JTextField( 30 );
      despliega.setEditable( false );
      c.add( despliega );
   } // fin del constructor VentanaPruebaHora

   public void despliegaHora()
   {
      despliega.setText( "La hora es: " + h );
   } // fin del método despliegaHora

   public static void main( String args[] )
   {
      VentanaPruebaHora ventana = new VentanaPruebaHora();

      ventana.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener

      ventana.setSize( 400, 120 );
      ventana.show();
   } // fin de main
} // fin de la clase VentanaPruebaHora
```

**Figura 27.7** Demostración de las clases anónimas internas; **`VentanaPruebaHora.java`**. (Parte 2 de 3).

Cuadro **Cerrar**

**Figura 27.7** Demostración de las clases anónimas internas; **`VentanaPruebaHora.java`**. (Parte 3 de 3).

Cada una de las tres **`JTextField`** que generan los eventos en este programa tiene una clase interna anónima para manipular los eventos, de modo que aquí solamente explicaremos la clase interna anónima para **`campoHora`**. Las líneas 24 a 34

```
campoHora.addActionListener(
   new ActionListener() {  // clase interna anónima
      public void actionPerformed( ActionEvent e )
      {
         h.estableceHora(
            Integer.parseInt( e.getActionCommand() ) );
         campoHora.setText( “” );
         despliegaHora();
      } // fin del método actionPerformed
   } // fin de la clase interna anónima
); // fin de addActionListener
```

llama al método **`campoHora`** de **`addActionListener`**. El argumento para este método debe ser un objeto que *es un* **`ActionListener`** (es decir, cualquier objeto de la clase que implementa **`ActionListener`**). Las líneas 25 a 33 utilizan una sintaxis especial de Java para definir una clase anónima interna y crear un objeto de la clase que se pasa como el argumento de **`ActionListener`**. La línea 25

```
new ActionListener() {   // clase interna anónima
```

utiliza el operador **`new`** para crear un objeto. La sintaxis **`ActionListener()`** inicia la definición de una clase interna anónima que implementa la interfaz **`ActionListener`**. Esto es similar a iniciar la definición de la clase como

```
public class MiManipulador implements ActionListener {
```

Los paréntesis después de **ActionListener** indican una llamada al constructor predeterminado de la clase anónima interna.

La llave izquierda de apertura (**{**) al final de la línea 25 y la llave derecha de cierre (**}**) en la línea 33 definen el cuerpo de la clase. Las líneas 26 a 32 definen el método, **actionPerformed**, que se requiere en cualquier clase que implementa **ActionListener**. Se llama al método **ActionPerformed** cuando el usuario presiona *Entrar* mientras escribe en **campoHora**.

### Observación de ingeniería de software 27.27

*Cuando una clase anónima interna implementa una interfaz, la clase debe definir cada método en la interfaz.*

El método **main** crea una instancia de la clase **VentanaPruebaHora** (línea 82), redimensiona la ventana (línea 93) y despliega la ventana (línea 94).

Windows genera distintos eventos que explicaremos en el capítulo 29. Para este ejemplo explicamos el evento generado cuando el usuario hace clic en el cuadro cerrar de la ventana, un *evento para cerrar la ventana*. Las líneas 84 a 91

```
ventana.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
```

permite al usuario terminar la aplicación al hacer clic en el cuadro Cerrar de la ventana (etiquetado en la primera pantalla de captura). El método ***addWindowListener*** registra un receptor de eventos de la ventana. El argumento **addWindowListener** debe ser una referencia a un objeto que *es un* **WindowListener** (paquete **java.awt.event**) (es decir, cualquier objeto de la clase que implementa **WindowListener**). Sin embargo, existen siete métodos diferentes que se deben definir en cada clase que implementa **WindowListener** y solamente necesitamos uno en este ejemplo, **WindowClosing**. Para las interfaces con más de un método, Java proporciona una clase correspondiente (llamada *clase adaptadora*) que de antemano implementa todos los métodos en la interfaz para usted. Todo lo que necesita hacer es extender la clase adaptadora y redefinir los métodos requeridos en su programa.



### Error común de programación 27.10

*Extender una clase adaptadora y escribir incorrectamente el nombre de un método que va a redefinir, es un error de lógica.*

Las líneas 85 a 90 utilizan una sintaxis especial de Java para definir una clase interna anónima y crear un objeto de la clase que se pasa como el argumento de **addWindowListener**. La línea 85

```
new WIndowAdapter() {
```

utiliza el operador **new** para crear un objeto. La sintaxis de **WindowAdapter()** comienza la definición de la clase que extiende a la clase **WindowAdapter**. Esto es similar al inicio de la definición de la clase

```
public class MiManipulador extiende WindowAdapter {
```

El paréntesis después de **WindowAdapter** indica una llamada al constructor predeterminado de la clase anónima interna. La clase **WindowAdapter** implementa la interfaz **WindowListener**, el tipo exacto requerido para el argumento de **addWindowListener.**

La llave izquierda de cierre (**{**) al final de la línea 85 y la llave derecha de cierre (**}**) en la línea 90 definen el cuerpo de la clase. Las líneas 86 a 89 redefinen el método de **WindowAdapter**, **windowClosing**, que se llama cuando el usuario hace clic en el cuadro Cerrar de la ventana. En este ejemplo, **windowClosing** termina la aplicación con una llamada a **System.exit(0**).

En los dos últimos ejemplos, vimos que las clases internas se pueden utilizar para crear manipuladores de eventos, y que las clases internas anónimas pueden definirse para manejar eventos de manera individual para cada componente GUI. En el capítulo 29, volveremos a revisar este concepto conforme expliquemos con detalle el mecanismo de manipulación de eventos.

## 27.18 Notas sobre las definiciones de clases internas

Esta sección presenta diversas notas de interés para los programadores con respecto a la definición y el uso de clases internas.

1. Compilar una clase que contiene clases internas da como resultado archivos separados **`.class`** para cada clase. Las clases internas con nombres tienen el nombre de archivo *NombreClaseExterna$NombreClaseInterna*. **`.class`**. Las clases internas anónimas tienen el nombre de archivo *NombreClaseExterna$#*. **`.class`**, donde # comienza en 1 y se incrementa para cada clase anónima que se encuentre durante la compilación.
2. Las clases internas con nombres de clases pueden definirse como **`public`**, **`protected`**, de acceso a paquetes o **`private`**, y están sujetas a las mismas restricciones de uso que los otros miembros de una clase.
3. Para acceder a la referencia **`this`** de una clase externa, utilice *NombreClaseExterna***`.this`**.
4. La clase externa es responsable de crear objetos de sus clases internas. Para crear un objeto de otra clase interna de la clase, primero genere un objeto de la clase externa y asígnelo a una referencia (a la que llamaremos **`ref`**). Después utilice una instrucción de la siguiente forma para crear un objeto de clase interna:

   *NombreClaseExterna*.*NombreClaseInterna* **`innerRef`** = **`ref.`**new *NombreClaseInterna*();
5. Una clase interna puede declararse como **`static`**. Una clase interna **`static`** no requiere que se defina un objeto de su clase externa (mientras que una clase interna no estática sí lo necesita). Una clase interna **`static`** no tiene acceso a los miembros no estáticos de la clase externa.

## 27.19 Clases envolventes para tipos primitivos

Cada uno de los tipos primitivos tiene una *clase de tipo envolvente*. A estas clases se les conoce como **`Character`**, **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`** y **`Boolean`**. Cada clase de tipo envoltura le permite manipular tipos primitivos como objetos de la clase **`Object`**. Por lo tanto, los valores de tipos de datos primitivos pueden procesarse de manera polimórfica, si se mantienen como objetos de clases de tipo envoltura. Muchas de las clases que desarrollaremos o que reutilizaremos manipulan y comparten objetos. Estas clases no pueden manipular de manera polimórfica variables de tipos primitivos, pero pueden manipular de manera polimórfica objetos de las clases de tipo envoltura, ya que en última instancia, toda clase se deriva de la clase **`Object`**.

Cada una de las clases numéricas (**`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`** y **`Double`**) hereda de la clase **`Number`**. Cada uno de los tipos de envoltura se declara **`final`**, por lo que sus métodos son implícitamente **`final`** y no pueden redefinirse. Observe que muchos de los métodos que procesan los tipos de datos primitivos se definen como métodos **`static`** de las clases de tipo envoltura. Si necesita manipular un valor primitivo en su programa, primero revise la documentación para las clases de tipo envoltura; es posible que el método que necesita ya esté definido.

## *RESUMEN*

- Una de las claves del poder de la programación orientada a objetos es lograr la reutilización de software a través de la herencia.
- A través de la herencia, una nueva clase hereda las variables y los métodos de instancia de una superclase previamente definida. En este caso, a la nueva clase se le conoce como subclase.
- Con herencia simple, una clase se deriva de una superclase. Con herencia múltiple, una subclase hereda de muchas superclases. Java no soporta la herencia múltiple, pero proporciona la idea de las interfaces, la cual ofrece muchos de los beneficios de la herencia múltiple sin los problemas asociados.

- Una subclase normalmente agrega variables y métodos de instancia por sí misma, por lo que una subclase generalmente es más grande que su superclase. Una subclase es más específica que su superclase, y normalmente representa pocos objetos.
- Una subclase no puede acceder a los miembros **`private`** de su superclase. Sin embargo, una subclase accede a los miembros **`public`**, **`protected`** y de acceso a paquetes de su superclase; la subclase debe estar en el paquete de la superclase para utilizar a los miembros de la superclase con acceso a paquetes.
- La herencia permite la reutilización de software, la cual ahorra tiempo de desarrollo y motiva el uso de software de alta calidad previamente probado y depurado.
- Algún día, la mayoría del software se construirá a partir de componentes reutilizables estandarizados, exactamente de la misma manera en que actualmente se hace la mayoría del hardware.
- Un objeto de una subclase puede tratarse como un objeto de su superclase correspondiente, pero lo contrario no es verdad.
- Una superclase existe en una relación jerárquica con sus subclases.
- Cuando una clase se utiliza con el mecanismo de la herencia, se vuelve una superclase que proporciona atributos y comportamientos a otras clases, o la clase se vuelve una subclase que hereda dichos atributos y comportamientos.
- Una jerarquía de herencia puede ser arbitrariamente profunda dentro de las limitaciones físicas de un sistema en particular, pero la mayoría de las jerarquías de herencia tienen sólo unos cuantos niveles.
- Las jerarquías son útiles para comprender y manejar la complejidad del software. Debido a que el software se vuelve cada vez más complejo, Java proporciona mecanismos para soportar estructuras jerárquicas a través de la herencia y el polimorfismo.
- El acceso **`protected`** sirve como un nivel intermedio de protección entre el acceso **`public`** y el **`private`**. A los miembros **`protected`** de una superclase pueden acceder los métodos de la superclase, los métodos de las subclases y los métodos de las clases en el mismo paquete; ningún otro método puede acceder a los miembros **`protected`** de una superclase.
- Una superclase puede ser una superclase directa de una subclase, o una superclase indirecta de una subclase. Una superclase directa es la clase que una subclase explícitamente amplía por medio de **`extends`**. Una superclase indirecta hereda de muchos niveles superiores en el árbol de jerarquía de clase.
- Cuando un miembro de una superclase es inadecuado para una subclase, el programador puede redefinir ese miembro en la subclase.
- Es importante diferenciar entre las relaciones *es un* y *tiene un*. En una relación *tiene un*, un objeto de una clase tiene como miembro a una referencia hacia un objeto de otra clase. En una relación *es un*, un objeto de un tipo de subclase también puede tratarse como un objeto de un tipo de superclase. *Es un* es herencia. *Tiene un* es composición.
- Un objeto de una subclase puede asignarse a una referencia de una superclase. Este tipo de asignación tiene sentido debido a que la subclase tiene miembros que corresponden a cada miembro de la superclase.
- Una referencia a un objeto de una subclase puede convertirse implícitamente en una referencia para un objeto de una superclase.
- Es posible convertir una referencia de una superclase en una referencia de una subclase por medio de una conversión de tipo explícita. Si el objetivo no es un objeto de una subclase, se lanza una **`ClassCastException`**.
- Una superclase especifica similitudes. Todas las clases derivadas de una superclase heredan las capacidades de esa superclase. En el proceso de diseño orientado a objetos, el diseñador busca similitudes entre clases y factores que toma para formar superclases. Las subclases entonces se personalizan más allá de las capacidades heredadas de la superclase.
- Leer un conjunto de declaraciones de subclases puede resultar confuso, ya que los miembros heredados de una superclase no se listan en las declaraciones de la subclase, pero estos miembros están realmente presentes en las subclases.
- Con el polimorfismo, se vuelve posible diseñar e implementar sistemas que son más fácilmente extensibles. Los programas pueden escribirse para procesar objetos de tipos que pueden no existir cuando el programa está en desarrollo.
- La programación polimórfica puede eliminar la necesidad de la lógica de **`switch`**, con lo que se evitan los tipos de errores asociados con dicha lógica.
- Un método abstracto se declara en la superclase precediendo la definición del método con la palabra reservada **`abstract`**.
- Existen muchas situaciones en las que es útil definir clases para las que el programador nunca intenta instanciar objeto alguno. Tales clases se conocen como clases **`abstract`**. Éstas se utilizan sólo como superclases, por lo que normalmente nos referiremos a ellas como superclases **`abstract`**. Ningún objeto de una clase **`abstract`** puede instanciarse.
- A las clases cuyos objetos pueden instanciarse se les conoce como clases concretas.

- Una clase se hace abstracta declarándola con la palabra reservada **`abstract`**.
- Si una subclase se deriva de una superclase con un método **`abstract`** sin proporcionar una definición para ese método **`abstract`** en la subclase, ese método permanece como **`abstract`** en la subclase. Como consecuencia, la subclase también es una clase **`abstract`** (y no puede instanciar objeto alguno).
- Cuando se hace una solicitud a través de una referencia de superclase para utilizar un método, Java elige el método redefinido correcto en la subclase asociada con el objeto.
- A través del polimorfismo, una llamada a un método puede ocasionar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada.
- Aunque no podemos crear instancias de objetos de superclases **`abstract`**, podemos declarar referencias hacia superclases **`abstract`**. Tales referencias pueden entonces utilizarse para permitir manipulaciones polimórficas de objetos de subclases, cuando dichos objetos son instanciados desde clases concretas.
- Con regularidad se agregan nuevas clases a los sistemas. Las nuevas clases se acomodan por medio del método de vinculación dinámica (también conocido como vinculación tardía). El tipo de un objeto no necesita conocerse en tiempo de compilación, para que se compile una llamada a un método. En tiempo de ejecución, se selecciona el método apropiado para recibir al objeto.
- Con el método de vinculación dinámica, en tiempo de ejecución, la llamada a un método se envía hacia la versión adecuada del método para la clase del objeto que recibe la llamada.
- Cuando una superclase proporciona un método, las subclases pueden redefinir el método, pero no tienen que hacerlo. Entonces, una subclase puede utilizar una versión de superclase de un método.
- Una definición de interfaz comienza con la palabra reservada **`interface`**, y contiene un conjunto de métodos **`public abstract`**. Las interfaces también pueden contener datos **`public final static`**.
- Para utilizar una interfaz, debe especificarse una clase que la implemente, y esa clase debe definir cada método en la interfaz con el número de argumentos y el tipo de retorno especificado en la definición de la interfaz.
- Por lo general, una interfaz se utiliza en lugar de una clase **`abstract`**, cuando no existe una implementación predeterminada a heredar.
- Cuando una clase implementa una interfaz, aplica la misma relación *es un* provista por la herencia.
- Para implementar más de una interfaz, en la definición de la clase simplemente proporcione una lista separada por comas con los nombres de las interfaces, después de la palabra reservada **`implements`**.
- Las clases internas se definen dentro del alcance de otras clases.
- Una clase interna también puede definirse dentro de un método de una clase. Tales clases internas tienen acceso a los miembros externos de la clase y a las variables locales **`final`** del método en el que están definidas.
- Las definiciones de clases internas se utilizan principalmente para la manipulación de eventos.
- La clase **`JFrame`** proporciona los atributos y comportamientos básicos de una ventana; una barra de título y botones para minimizar, maximizar y cerrar la ventana.
- Un objeto de una clase interna tiene una relación especial con el objeto de la clase externa que lo crea. Al objeto de la clase interna se le permite acceder directamente a todas las variables y métodos de instancia del objeto de la clase externa.
- Una clase interna anónima no tiene nombre, por lo que un objeto de una clase interna anónima debe crearse en el punto en el que la clase se define en el programa.
- Una clase interna anónima puede implementar una interfaz o extender una clase.
- El evento generado cuando el usuario hace clic en el cuadro **Close** de la ventana, es un evento de cierre de ventana.
- El método **`addWindowListener`** registra un oyente del evento ventana. Su argumento debe ser una referencia hacia un objeto que es un **`WindowListener`** (paquete **`java.awt.event`**).
- Para interfaces de manejo de eventos con más de un método, Java proporciona una clase correspondiente (llamada clase adaptadora) que implementa para usted todos los métodos en la interfaz. La clase **`WindowAdapter`** implementa la interfaz **`WindowListener`**, de tal forma que todo objeto **`WindowAdapter`** *es un* **`WindowListener`**.
- Compilar una clase que contiene clases internas da como resultado un archivo **`.class`** para cada clase.
- Las clases internas con nombres de clases pueden definirse como **`public`**, **`protected`**, de acceso a paquetes o **`private`**, y están sujetas a las mismas restricciones de uso que los otros miembros de una clase.
- Para acceder a la referencia **`this`** de una clase externa, utilice *NombreClaseExterna***`.this`**.
- La clase externa es responsable de crear objetos de sus clases internas no estáticas.
- Una clase interna puede declararse como **`static.`**

## TERMINOLOGÍA

abstracción
clase **abstract**
clase base
clase **Boolean**
clase **Character**
clase **Double**
clase envolvente
clase **final**
clase **Integer**
clase interna
clase interna anónima
clase **JFrame**
clase **Long**
clase **Number**
clase **Object**
clase **WindowAdapter**
clase **WindowEvent**
cliente de una clase
componentes de software estandarizados
composición
constructor de una subclase
constructor de una superclase
control de acceso a miembros
conversión implícita de referencia
**extends**
extensibilidad
herencia
herencia de implementación
herencia de interfaz
herencia múltiple
herencia simple
interfaz
interfaz **WindowListener**
jerarquía de clase
jerarquía de herencia
lógica **switch**
método **abstract**
método de vinculación dinámica
método **final**
método **show**
método **windowClosing**
miembro **protected** de una clase
objeto miembro
polimorfismo
programación orientada a objetos (**POO**)
recolector de basura
redefinición *vs* sobrecarga
redefinir un método
redefinir un método **abstract**
referencia hacia una clase **abstract**
referencia hacia una subclase
referencia hacia una superclase
relación *es un*
relación jerárquica
relación *tiene un*
reutilización de software
subclase
**super**
superclase
superclase **abstract**
superclase directa
superclase indirecta
**this**
variable de instancia **final**
vinculación tardía

## ERRORES COMUNES DE PROGRAMACIÓN

**27.1** Tratar a un objeto de una superclase como un objeto de una subclase puede ocasionar errores.

**27.2** Asignar un objeto de una superclase a una referencia de una subclase (sin una conversión de tipo), es un error de sintaxis.

**27.3** Si una subclase hace una llamada **super** al constructor de su superclase, y esta llamada no es la primera instrucción en el constructor de la subclase, es un error de sintaxis.

**27.4** Si los argumentos de una llamada **super** de una subclase al constructor de su superclase no coinciden con los parámetros especificados en una de las definiciones del constructor de la superclase, es un error de sintaxis.

**27.5** Si un método de una superclase y un método en su subclase tienen la misma firma pero diferente tipo de retorno, es un error de sintaxis.

**27.6** Asignar un objeto de subclase a una referencia de superclase, y después intentar hacer referencia sólo a miembros de la subclase con la referencia de superclase, es un error de sintaxis.

**27.7** Intentar crear una instancia de un objeto de una clase abstracta (es decir, una clase que contiene uno o más métodos abstractos), es un error de sintaxis.

**27.8** Si una clase con uno o más métodos **abstract** no se declara específicamente como **abstract**, es un error de sintaxis.

**27.9** Dejar indefinido un método de una interfaz, en una clase que implementa la interfaz, da como resultado un error de compilación que indica que la clase debe declararse como **abstract**.

**27.10** Extender una clase adaptadora y escribir incorrectamente el nombre de un método que va a redefinir, es un error de lógica.

## TIPS PARA PREVENIR ERRORES

**27.1** Ocultar los miembros **private** es una gran ayuda al probar, depurar y modificar correctamente los sistemas. Si una subclase pudiera acceder a los miembros **private** de su superclase, entonces sería posible que las clases derivadas de esa subclase accedieran también a esos datos, y así sucesivamente. Esto propagaría el acceso a lo que se supone deberían ser datos **private**, y los beneficios del ocultamiento de información se perderían a lo largo de la jerarquía de la clase.

**27.2** Una consecuencia interesante de utilizar el polimorfismo es que los programas adquieren una apariencia simplificada; contienen menos lógica de separación, a favor de un código secuencial más sencillo. Esta simplificación facilita el probar, depurar y mantener un programa.

## TIPS DE RENDIMIENTO

**27.1** Si las clases producidas a través de la herencia son más grandes de lo necesario, podrían desperdiciarse recursos de memoria y de procesamiento. Herede de la clase "que más se acerque" a lo que usted necesita.

**27.2** El compilador puede decidir poner en línea a una llamada a un método **`final`**, y lo hará para métodos **`final`** pequeños y sencillos. Colocarlas en línea no viola el encapsulamiento o el ocultamiento de información (pero mejora el rendimiento, ya que elimina la sobrecarga de realizar una llamada a un método).

**27.3** Los preprocesadores canalizados pueden mejorar el rendimiento ejecutando simultáneamente diversas partes de las siguientes instrucciones, pero no si esas instrucciones siguen a una llamada a un método. Colocar en línea al código (lo que el compilador realiza en un método **`final`**) puede mejorar el rendimiento de estos preprocesadores, ya que elimina la transferencia de control fuera de línea asociada con una llamada a un método.

**27.4** Cuando el polimorfismo se implementa con el método de vinculación dinámica, es eficiente.

**27.5** Los tipos de manipulaciones polimórficas que se hacen posibles con la vinculación dinámica, también pueden lograrse por medio de la lógica de **`switch`** codificada manualmente, de acuerdo con los campos de tipo de los objetos. El código polimórfico generado por el compilador de Java se ejecuta con un rendimiento comparable con la lógica de **`switch`** eficientemente codificada.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**27.1** Una subclase no puede acceder directamente a miembros **`private`** de su superclase.

**27.2** Los constructores nunca se heredan; éstos son específicos de la clase en la que están definidos.

**27.3** Si un objeto se ha asignado a una referencia de una de sus superclases, es aceptable convertir el tipo de ese objeto de regreso a su propio tipo. De hecho, esto debe hacerse para enviar a ese objeto cualquiera de los mensajes que no aparecen en esa superclase.

**27.4** Toda clase en Java extiende a **`Object`**, a menos que se especifique lo contrario en la primera línea de la definición de la clase. Por lo tanto, la clase **`Object`** es la superclase de toda la jerarquía de clases de Java.

**27.5** Una redefinición de un método de una superclase en una subclase no tiene la misma firma que el método de la superclase. Tal redefinición no es la redefinición de un método, sino un simple ejemplo de la sobrecarga de métodos.

**27.6** Cualquier objeto puede convertirse en una **`String`** con una llamada explícita o implícita al método **`toString`** del objeto.

**27.7** Toda clase debe redefinir el método **`toString`** para devolver información útil sobre los objetos de esa clase.

**27.8** Crear una subclase no afecta el código fuente de su superclase, o el código en bytes de las superclases de Java; la integridad de una superclase se preserva a través de la herencia.

**27.9** Así como el diseñador de sistemas no orientados a objetos deben evitar la proliferación de funciones innecesarias, el diseñador de sistemas orientados a objetos debe evitar la proliferación de clases innecesarias. La proliferación de clases genera problemas de administración y puede dificultar la reutilización de software, simplemente porque es más difícil para un usuario potencial de una clase localizar esa clase en una amplia colección. El equilibrio se encuentra en crear pocas clases que proporcionen funcionalidad adicional importante, sin embargo, dichas clases pueden ser demasiado ricas para ciertos usuarios.

**27.10** En un sistema orientado a objetos, con frecuencia las clases se encuentran muy relacionadas. "Ubique" los atributos y comportamientos comunes y colóquelos en una superclase. Después utilice la herencia para formar subclases para que no tenga que repetir atributos y comportamientos comunes.

**27.11** Las modificaciones a una superclase no requieren que las subclases se modifiquen, mientras la interfaz pública de la superclase permanezca sin cambios.

**27.12** Cuando una subclase elige no redefinir un método, la subclase simplemente hereda la definición del método de su superclase inmediata.

**27.13** Una clase definida como **`final`** no puede extenderse, y cada uno de sus métodos es implícitamente **`final`**.

**27.14** Una clase abstracta puede tener datos de instancia y métodos no abstractos sujetos a las reglas normales de la herencia de las subclases. Una clase abstracta también pueden tener constructores.

**27.15** Si una subclase se deriva de una superclase con un método **`abstract`**, y si no se proporciona una definición en la subclase para ese método **`abstract`** (es decir, si no se redefine ese método en la subclase), ese método permanece como **`abstract`** en la subclase. Como consecuencia, la subclase también es una clase **`abstract`**, y debe declararse explícitamente como **`abstract`**.

**27.16** La habilidad de declarar un método **`abstract`** le da al diseñador de la clase suficiente poder sobre cómo implementará las subclases en una jerarquía de clases. Cualquier clase nueva que quiera heredar de esta clase es forzada a redefinir el método **`abstract`** (ya sea directamente o heredando de una clase que ha redefinido el método). De lo contrario, esa nueva clase contendrá un método **`abstract`** y, por lo tanto, será una clase **`abstract`** incapaz de instanciar objetos.

**27.17** Con el polimorfismo, el programador puede lidiar con las generalidades y deja que el ambiente en tiempo de ejecución se ocupe de lo específico. El programador puede ordenar que una amplia variedad de objetos se comporten de manera apropiada sin siquiera conocer los tipos de esos objetos.

**27.18** El polimorfismo promueve la extensibilidad: El software escrito para invocar un comportamiento polimórfico se escribe de manera independiente a los tipos de los objetos a los que se envían los mensajes (es decir, llamadas a métodos). Por lo tanto, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en tales sistemas sin modificar el sistema base.

**27.19** Si un método se declara como **`final`**, éste no puede redefinirse en las subclases, por lo que las llamadas al método no pueden enviarse de manera polimórfica a los objetos de esas subclases. La llamada al método aún puede enviarse a las subclases, pero responderán de manera idéntica, en lugar de hacerlo de manera polimórfica.

**27.20** Una clase **`abstract`** define una interfaz común para los diversos miembros de una jerarquía de clase. La clase **`abstract`** contiene métodos que se definirán en las subclases. Todas las clases de la jerarquía pueden utilizar esta misma interfaz a través del polimorfismo.

**27.21** Las jerarquías diseñadas para la herencia de la implementación tienden a tener a su funcionalidad arriba en la jerarquía; cada nueva subclase hereda uno o más de los métodos que se definieron en una superclase, y utiliza las definiciones de la superclase.

**27.22** Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad más abajo en la jerarquía; una superclase especifica uno o más métodos que deben invocarse de manera idéntica para cada objeto en la jerarquía (es decir, tienen la misma firma), pero las subclases individuales proporcionan sus propias implementaciones de los métodos.

**27.23** Una subclase siempre hereda la versión definida más recientemente de cada método **`public`** y **`protected`** de sus superclases directa e indirecta.

**27.24** Todos los métodos de la clase **`Object`** pueden invocarse por medio de una referencia a un tipo de dato interfaz; una referencia se refiere a un objeto, y todos los objetos tienen los métodos definidos por la clase **`Object`**.

**27.25** A un objeto de la clase interna se le permite tener acceso directo a las variables de instancia y a los métodos del objeto de la clase externa que la define.

**27.26** Una clase interna definida dentro de un método tiene acceso directo a todas las variables de instancia y métodos del objeto de la clase externa en la que se define y en cualquier variable local de **`final`** en el método.

**27.27** Cuando una clase anónima interna implementa una interfaz, la clase debe definir cada método en la interfaz.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**27.1** Complete los espacios en blanco:

a) Si la clase **`Alfa`** hereda de la clase **`Beta`**, a la clase **`Alfa`** se le conoce como ___________, y a la clase **`Beta`** se le conoce como ___________.
b) La herencia permite la ___________, la cual ahorra tiempo de desarrollo y motiva el uso de componentes de software de alta calidad previamente probados.
c) Un objeto de una clase puede tratarse como un objeto de su ___________correspondiente.
d) Los cuatro especificadores de acceso a miembros son __________, __________, __________ y __________.
e) Una relación *tiene un* entre clases representa a la ___________, y una relación *es un* entre clases representa a la ___________.
f) Utilizar el polimorfismo ayuda a eliminar la lógica de ___________.
g) Si una clase contiene uno o más métodos **`abstract`**, se trata de una clase ___________.
h) Una llamada a un método resuelta en tiempo de ejecución se conoce como vinculación ___________.

**27.2** a) Una subclase puede llamar a cualquier método de una superclase no **`private`**, anteponiendo ___________ a la llamada al método.
b) Una superclase generalmente representa a un número mayor de objetos que su subclase. (*Verdadero/falso*.)
c) Una subclase normalmente encapsula menos funcionalidad que su superclase. (*Verdadero/falso*.)

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**27.1** a) Subclase, superclase. b) Reutilización de software. c) Subclase, superclase. d) **`public`**, **`protected`**, **`private`** y de acceso a paquetes. e) Composición, herencia. f) **`switch`**. g) **`abstract`**. h) Dinámica.

**27.2** a) **`super`**
b) verdadero
c) falso

## EJERCICIOS

**27.3** Considere la clase **`Bicicleta`**. Dado su conocimiento sobre algunos componentes de bicicletas, muestre una jerarquía en la que la clase **`Bicicleta`** herede de otras clases, las cuales, a su vez, hereden de otras clases. Explique la creación de instancias de varios objetos de la clase **`Bicicleta`**. Explique la herencia de la clase **`Bicicleta`** para otras subclases muy relacionadas.

**27.4** Defina cada uno de los siguientes términos: herencia simple, herencia múltiple, interfaz, superclase y subclase.

**27.5** Explique por qué convertir el tipo de una referencia de superclase en una referencia de subclase es potencialmente peligroso.

**27.6** Plantee las diferencias entre la herencia simple y la herencia múltiple. ¿Por qué Java no soporta la herencia múltiple? ¿Qué característica de Java ayudan a contar con los beneficios de la herencia múltiple?

**27.7** (*Verdadero*/*Falso*.) Una subclase es generalmente más pequeña que su superclase.

**27.8** (*Verdadero*/*Falso*.) Un objeto de una subclase es también un objeto de la superclase de esa subclase.

**27.9** Rescriba el programa **`Punto`**, **`Circulo`**, **`Cilindro`** de la figura 27.4 como un programa **`Punto`**, **`Cuadrado`**, **`Cubo`**. Haga esto de dos formas: una con herencia y otra con composición.

**27.10** En el capítulo dijimos que "cuando un método de una superclase es inadecuado para una subclase, ese método puede redefinirse en la subclase con una implementación adecuada". Si se hace esto, ¿la relación "el objeto de una subclase es un objeto de la superclase", se mantiene? Explique su respuesta.

**27.11** ¿Cómo es que el polimorfismo le permite programar "en general", en lugar de "en específico"? Explique las principales ventajas de la programación "en general".

**27.12** Explique los problemas de la programación con lógica de **`switch`**. Explique por qué el polimorfismo es una alternativa efectiva al uso de la lógica de **`switch`**.

**27.13** Plantee la diferencia entre herencia de interfaz y herencia de implementación. ¿Cómo difieren las jerarquías de herencia diseñadas para herencia de interfaz de aquellas diseñadas para herencia de implementación?

**27.14** Plante la diferencia entre métodos no abstractos y los métodos abstractos.

**27.15** (*Verdadero*/*Falso*.) Todos los métodos de una superclase **`abstract`** deben declararse como **`abstract`**.

**27.16** Sugiera uno o más niveles de superclases **`abstract`** para la jerarquía **`Figura`** que explicamos al principio de este capítulo (el primer nivel es **`Figura`**, y el segundo nivel consiste en las clases **`FiguraBidimensional`** y **`FiguraTridimensional`**).

**27.17** ¿Cómo es que el polimorfismo promueve la extensibilidad?

**27.18** Se le ha pedido que desarrolle un simulador de vuelo que tendrá que elaborar resultados gráficos. Explique por qué la programación polimórfica sería especialmente efectiva para un problema de esta naturaleza.

**27.19** (*Aplicación de dibujo*.) Modifique el programa de dibujo del ejercicio 26.11 para crear una aplicación de dibujo que dibuje líneas aleatorias, rectángulos y óvalos. [*Nota:* Como un applet, **`JFrame`** tiene un método **`paint`** que puede redefinir para dibujar en el fondo del **`JFrame`**.]

Para este ejercicio, modifique las clases **`MiLinea`**, **`MiElipse`** y **`MiRectangulo`** del ejercicio 26.11 para crear la jerarquía de clase de la figura 27.8. Las clases de la jerarquía **`MiFigura`** deben ser clases de figuras "inteligentes", en donde los objetos de estas clases sepan cómo dibujarse a sí mismas (si cuentan con un objeto **`Graphics`** que les indique dónde dibujar). La única lógica de **`switch`** o de **`if/else`** en este programa debe ser

para determinar el tipo de objeto figura a crear (utilice números aleatorios para escoger el tipo de figura y las coordenadas de cada figura.) Una vez que se cree un objeto de esta jerarquía, éste será manipulado por el resto de su tiempo de vida como una referencia de la superclase **`MiFigura`**.

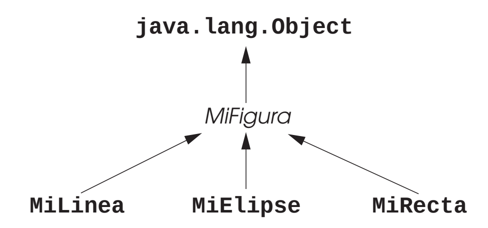



**Figura 27.8** Jerarquía **`MiFigura`**.

La clase **`MiFigura`** de la figura 27.8 *debe* ser **`abstract`**. El único dato que representa las coordenadas de las figuras de la jerarquía debe definirse en la clase **`MiFigura`**. Las líneas, los rectángulos y las elipses pueden dibujarse si conoce dos puntos en el espacio. Las líneas requieren coordenadas *x1*, *y1* y *x2*, *y2*. El método **`drawLine`** de la clase **`Graphics`** conectará con una línea los dos puntos proporcionados. Si usted tiene los mismos cuatro valores para las coordenadas (*x1*, *y1* y *x2*, *y2*) para elipses y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere un valor para la coordenada superior izquierda *x* (el mínimo de los dos valores para las coordenadas en *x*), un valor para la coordenada superior izquierda y (el mínimo de los dos valores para las coordenadas en *y*), un *ancho* (la diferencia entre los dos valores correspondientes a las coordenadas en *x*; la cual debe ser positiva) y una *altura* (la diferencia entre los dos valores correspondientes a las coordenadas en *y*; la cual debe ser positiva). [*Nota:* En el capítulo 29, cada par *x*,*y* se capturará utilizando eventos del ratón, a partir de interacciones del ratón entre el usuario y el fondo del programa. Estas coordenadas se almacenarán en el objeto de figura adecuado, conforme seleccione el usuario. Conforme inicie el ejercicio, utilizará valores aleatorios para las coordenadas como argumentos del constructor.]

Además de los datos para la jerarquía, la clase **`MiFigura`** debe definir al menos los siguientes métodos:

a) Un constructor sin argumentos que establezca en cero a las coordenadas.
b) Un constructor con argumentos que establezca las coordenadas en los valores proporcionados.
c) Métodos *establecer* para cada pieza individual de datos que permita al programador establecer de manera independiente cualquier pieza de datos para una figura de la jerarquía (por ejemplo, si tiene una variable de instancia **`x1`**, debe tener un método **`estableceX1`**).
d) Métodos *obtener* para cada pieza individual de datos que permita al programador recuperar de manera independiente cualquier pieza de datos para una figura de la jerarquía (por ejemplo, si tiene una variable de instancia **`x1`**, debe tener un método **`obtieneX1`**).
e) El método **`abstact`**

```
public abstract void draw( Graphics g );
```

Este método será llamado desde el método **`paint`** del programa para dibujar una figura en la pantalla.

Los métodos anteriores son necesarios. Si quisiera proporcionar más métodos para una mayor flexibilidad, hágalo. Sin embargo, asegúrese de que cualquier método que defina en esta clase sea un método que se utilizará en *todas* las figuras de la jerarquía.

Todos los datos deben ser **`private`** para la clase **`MiFigura`** de este ejercicio (esto lo obliga a utilizar el encapsulamiento de datos adecuado, y a proporcionar los métodos *establecer/obtener* adecuados para manipular los datos). No se le permite definir nuevos datos que puedan derivarse de información existente. Como explicamos anteriormente, la *x* superior izquierda, la *y* superior izquierda, el *ancho* y la *altura* son necesarios para dibujar un óvalo o para calcular un rectángulo, si usted ya conoce dos puntos en el espacio. Todas las subclases de **`MiFigura`** deben proporcionar dos constructores que imiten a los proporcionados por la clase **`MiFigura`**.

Los objetos de las clases **`MiElipse`** y **`MiRecta`** no deben calcular sus coordenadas superiores izquierdas ***`x`*** y ***`y`***, y el *ancho* y la *altura*, hasta que se vayan a dibujar. Nunca modifique las coordenadas *x1*, *y1* y *x2 y y2* de un objeto **`MiElipse`** o **`MiRecta`** para prepararse a dibujarlos. En su lugar, utilice resultados temporales de los cálculos descritos arriba. Esto nos ayudará a mejorar el programa del capítulo 29, que permitirá al usuario seleccionar con el ratón las coordenadas de cada figura.

En el programa no debe haber referencias **`MiLinea`**, **`MiElipse`** o **`MiRecta`**; sólo están permitidas las referencias de **`MiFigura`** que hagan referencia a objetos **`MiLinea`**, **`MiElipse`** y **`MiRecta`**.

El programa debe mantener un arreglo de referencias **MiFigura** que contenga todas las figuras. El método **paint** del programa deben recorrer el arreglo de referencias **MiFigura** y dibujar todas las figuras (es decir, llamar a cada método **draw** de las figuras).

Comience definiendo la clase **MiFigura**, la clase **MiLinea** y una aplicación para probar sus clases. La aplicación debe tener una variable de instancia que pueda referirse a un objeto **MiLinea** (creado en el constructor de la aplicación). El método **paint** (para su subclase **JFrame**) debe dibujar la figura con una instrucción como

```
figuraActual.draw( g );
```

donde **figuraActual** es la referencia **MiFigura** y **g** es el objeto **Graphics** que la figura utilizará para dibujarse a sí misma en el fondo de la ventana.

Después, cambie la referencia simple **MiFigura** hacia un arreglo de referencias de **MiFigura**, y codifique diversos objetos **MiLinea** en el programa de dibujo. El método **paint** de la aplicación debe recorrer el arreglo de figuras y dibujar cada figura.

Después de que la parte anterior esté funcionando, debe definir las clases **MiElipse** y **MiRecta**, y agregar objetos de estas clases en el arreglo existente. Por el momento, todos los objetos de figuras deben crearse en el constructor de su subclase **JFrame**. En el capítulo 29, crearemos los objetos cuando el usuario elija una figura y comience a dibujarlo con el ratón.



# Gráficos en Java y Java2D




## Objetivos

- Comprender los contextos y los objetos gráficos.
- Comprender y manipular colores.
- Comprender y manipular fuentes.
- Comprender y utilizar los métodos de **`Graphics`** para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, rectángulos de tres dimensiones, elipses, arcos y polígonos.
- Utilizar los métodos de la clase **`Graphics2D`** de la API **`Java2D`** para dibujar líneas, rectángulos, rectángulos con líneas redondeadas, elipses, arcos y patrones generales.
- Especificar las características **`Paint`** y **`Stroke`** de las figuras desplegadas con **`Graphics2D`**.

*Una imagen vale más que mil palabras.*
Proverbio Chino

*Trata a la naturaleza como a un cilindro, una esfera, un cono, todas en perspectiva.*
Paul Cézanne

*Nada es real hasta que se experimenta, incluso un proverbio no es proverbio hasta que la vida se lo ilustra.*
John Keats

*Una imagen me muestra al instante lo que a un libro le lleva docenas de páginas.*
Ivan Sergeyevich

## Plan general



## 28.1 Introducción

En este capítulo, echaremos un vistazo a las capacidades de Java para dibujar figuras de dos dimensiones, para controlar los colores y para controlar las fuentes. Uno de los atractivos iniciales de Java era el soporte para gráficos que permitía a los programadores mejorar visualmente sus applets y aplicaciones. Ahora, Java contiene muchas más capacidades sofisticadas que forman parte de la API *Java2D*. Este capítulo comienza con una introducción a muchas de las capacidades originales de Java. A continuación, presentamos varias de las nuevas y más poderosas capacidades de Java2D, tales como el control del estilo de las líneas que se utilizan para dibujar las figuras y el control para rellenar figuras con colores y patrones.

La figura 28.1 muestra una parte de la jerarquía de clases de Java que incluyen varias de las distintas clases para gráficos básicos y las clases de la API Java2D, así como las interfaces que hemos tratado en este libro. La clase **`Color`** contiene los métodos y las constantes para manipular colores. La clase **`Font`** contiene los métodos y las constantes para manipular fuentes. La clase **`FontMetrics`** contiene los métodos obtener la información de las fuentes. La clase **`Polygon`** contiene los métodos para crear polígonos. La clase **`Graphics`** contiene los métodos para dibujar cadenas, líneas, rectángulos y otras figuras. La mitad inferior de la figura lista varias clases e interfaces de la API Java2D. La clase **`BasicStroke`** ayuda a especificar las características de las líneas. Las clases **`GradientPaint`** y **`TexturePaint`** ayudan a especificar las características para el relleno de las figuras con colores y patrones. Las clases **`GeneralPath`**, **`Arc2D`**, **`Ellipse2D`**, **`Line2D`**, **`Rectagle2D`**, y **`RoundRectangle2D`** definen una variedad de figuras de **`Java2D`**.

Para comenzar a dibujar en Java, primero debemos comprender el *sistema de coordenadas* de Java (figura 28.2), el cual es un esquema para identificar cada posible punto en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente GUI (tal como un applet o una ventana) tiene las coordenadas (0,0). Un par de coordenadas está compuesto por una *coordenada x* (la *coordenada horizontal*) y una *coordenada y* (la *coordenada vertical*). La coordenada *x* es la distancia horizontal de movimiento hacia la derecha, desde la esquina superior izquierda. La coordenada *y* es la distancia vertical de movimiento hacia abajo, desde la esquina superior izquierda. El *eje x* describe cada coordenada horizontal, y el *eje y* describe cada coordenada vertical.



**Observación de ingeniería de software 28.1**

*La coordenada superior izquierda (0,0) de una ventana en realidad se encuentra debajo de la barra de título de la ventana. Por esta razón, las coordenadas de dibujo deben ajustarse para dibujar dentro de los bordes de la ventana. La clase* **`Container`** *(una superclase de todas las ventanas en Java) contiene el método* **`getInsets`** *que devuelve un objeto* **`Instets`** *(del paquete* **`java.awt`***) para este propósito. Un objeto* **`Insets`** *contiene cuatro miembros públicos,* **`top`**, **`bottom`**, **`left`** *y* **`right`**, *que representan el número de pixeles de cada borde de la ventana hacia el área de dibujo de ésta.*

El texto y las figuras se despliegan en la pantalla especificando las coordenadas. Las unidades de las coordenadas se miden en *pixeles*. Un *píxel* es la unidad de resolución más pequeña de la pantalla.

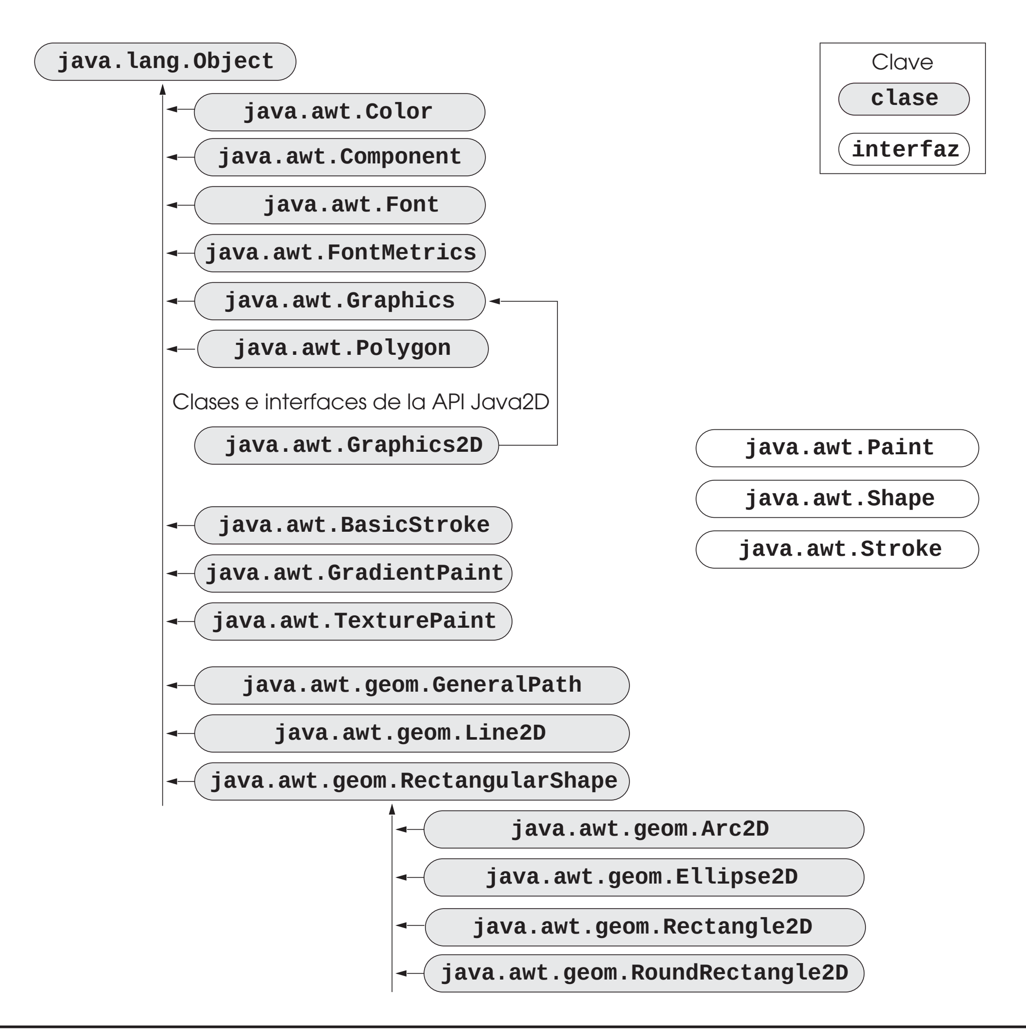



**Figura 28.1** Algunas clases e interfaces de las capacidades gráficas originales de Java y de la API Java2D, que utilizamos en este capítulo.

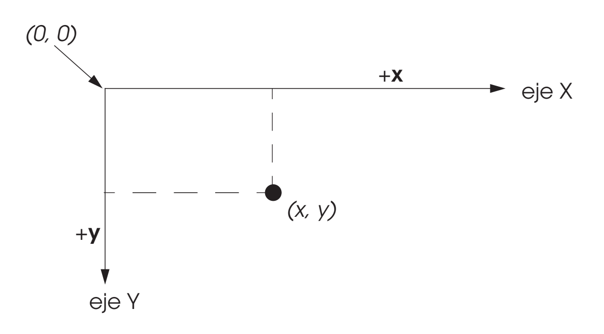



**Figura 28.2** Sistema de coordenadas de Java. Las unidades se miden en pixeles.



### Tip de portabilidad 28.1

*Diferentes pantallas tienen diferentes resoluciones (es decir, varía la densidad de pixeles). Esto puede provocar que los gráficos parezcan de tamaño diferente en diferentes pantallas.*

## 28.2 Contextos gráficos y objetos gráficos

Un *contexto gráfico* en Java permite dibujar en la pantalla. Un objeto de **Graphics** manipula un contexto gráfico al controlar la forma en que se dibuja en él. Los objetos de **Graphics** contienen métodos para dibujar, para manipular fuentes, para manipular colores y otros aspectos similares. Cada uno de los applets que vimos en el texto que realiza el dibujo en la pantalla utiliza en el objeto **g** de **Graphics** (el argumento para el método **paint** del applet) para manipular el contexto gráfico del applet. En este capítulo, mostraremos las aplicaciones de dibujo. Sin embargo, cada técnica mostrada puede ser útil en los applets.

La clase **Graphics** es una clase **abstract** (es decir, los objetos de **Graphics** no pueden instanciarse). Esto contribuye a la portabilidad de Java. El dibujo se realiza de manera diferente en cada plataforma que soporta Java, de modo que no puede existir una clase que implemente todas las capacidades de dibujo en un solo sistema. Por ejemplo, las capacidades gráficas que permiten a una PC que ejecuta Microsoft Windows dibujar un rectángulo, son diferentes de las capacidades que permiten a una estación de trabajo en UNIX dibujar el mismo rectángulo, y ambas son diferentes a las capacidades que permiten a una Macintosh dibujar un rectángulo. Cuando se implementa Java en cada plataforma, se crea una clase derivada de **Graphics** que en realidad implementa todas las capacidades de dibujo. Esta implementación se nos oculta por medio de la clase **Graphics**, la cual suministra la interfaz que nos permite escribir programas para utilizar gráficos de manera independiente de la plataforma.

La clase **Component** es la superclase de muchas de las clases en el paquete **java.awt** (explicaremos la clase **Component** en el capítulo 29). El método **paint** de **Component** toma un objeto **Graphics** como argumento. Este objeto se pasa al método **paint** mediante el sistema, cuando se requiere una operación **paint** para un **Componente**. El encabezado para el método **paint** es

```
public void paint( Graphics g )
```

El objeto **paint** recibe una referencia a un objeto de la clase derivada de **Graphics**. El método anterior debe parecerle conocido; es el mismo que hemos utilizado en nuestras clases de applets. En realidad, la clase **Component** es una clase base indirecta de la clase **JApplet**, la superclase de cada applet del libro. El método **paint** definido en la clase **Component** no hace cosa alguna de manera predeterminada, el programador la debe redefinir.

Por lo general, el programador llama directamente al método **paint**, debido a que dibujar los gráficos es un *proceso controlado por eventos*. Cuando se ejecuta un applet, al método **paint** se le llama automáticamente (después de las llamadas a los métodos **init** y **start**). Para poder llamar de nuevo a **paint**, debe ocurrir un *evento* (tal como cubrir o descubrir un applet). De manera similar, cuando se despliega un **Component**, se llama al método **paint** de dicho componente.

Si el programador necesita llamar a **paint**, se hace una llamada al método **repaint** de la clase **paint**. El método **repaint** solicita al usuario una llamada al método **update** de la clase **Component** tan pronto como sea posible limpiar cualquier dibujo previo, del fondo del componente, luego **update** llama directamente a **paint**. El programador llama con frecuencia al método **repaint** para forzar la operación **paint**. El método **repaint** no debe redefinirse debido a que realiza algunas tareas dependientes del sistema. Con frecuencia, al método **update** se le llama de manera directa y algunas veces se redefine. Redefinir el método **update** es útil para "suavizar" las animaciones (es decir, reducir las "asperezas") como explicaremos en el capítulo 30. Los encabezados para **repaint** y **update** son

```
public void repaint()
public void update( Graphics g )
```

El método **update** toma un objeto **Graphics** como argumento, el cual es suministrado automáticamente por el sistema cuando se llama a **update**.

En este capítulo, nos concentraremos en el método **paint**. En el siguiente capítulo nos concentraremos más en la naturaleza controlada por eventos de los gráficos y explicaremos los métodos **repaint** y **update** con más detalle. También explicaremos la clase **JComponent**, una superclase de muchos componentes GUI en el paquete **javax.swing**. Por lo general, las subclases de **JComponent** pintan a partir de los métodos **paint** de **Component**.

## 28.3 Control del color

Los colores mejoran la apariencia de un programa y ayudan a trasmitir su significado. Por ejemplo, una luz de semáforo tiene tres diferentes luces de colores, el rojo indica alto, el amarillo indica precaución y el verde indica adelante.

La clase **`Color`** define los métodos y las constantes para manipular los colores en un programa en Java. Las constantes para los colores predefinidos aparecen en la figura 28.3, y la figura 28.4 resume distintos métodos y constructores de colores. Observe que los dos métodos de la figura 28.4 son los métodos de **`Graphics`** que son específicos para los colores.

| Constante del color | Color | valor RGB (RVA) |
|---|---|---|
| `public final static Color orange` | naranja | 255, 200, 0 |
| `public final static Color pink` | rosa | 255, 175, 175 |
| `public final static Color cyan` | cian | 0, 255, 255 |
| `public final static Color magenta` | magenta | 255, 0, 255 |
| `public final static Color yellow` | amarillo | 255, 255, 0 |
| `public final static Color black` | negro | 0, 0, 0 |
| `public final static Color white` | blanco | 255, 255, 255 |
| `public final static Color gray` | gris | 128, 128, 128 |
| `public final static Color lightGray` | gris claro | 192, 192, 192 |
| `public final static Color darkGray` | gris oscuro | 64, 64, 64 |
| `public final static Color red` | rojo | 255, 0, 0 |
| `public final static Color green` | verde | 0, 255, 0 |
| `public final static Color blue` | azul | 0, 0, 255 |

**Figura 28.3** Constantes estáticas de la clase **`Color`** y valores RGB (RVA).

| Método | Descripción |
|---|---|
| `public Color( int r, int g, int b )` | |
| | Crea un color basado en contenido de rojo, verde y azul expresados como enteros desde 0 hasta 255. |
| `public Color( float r, float g, float b )` | |
| | Crea un color basado en contenido de rojo, verde y azul expresados como flotantes entre 0.0. y 1.0. |
| `public int getRed()            // clase Color` | |
| | Devuelve un valor entre 0 y 255 que representa el contenido de rojo. |
| `public int getGreen()          // clase Color` | |
| | Devuelve un valor entre 0 y 255 que representa el contenido de verde. |
| `public int getBlue()           // clase Color` | |
| | Devuelve un valor entre 0 y 255 que representa el contenido de azul. |
| `public Color getColor()        // clase Graphics` | |
| | Devuelve un objeto **`Color`** que representa el color actual para el contexto gráfico. |
| `public void setColor( Color c ) // clase Graphics` | |
| | Establece el color actual para dibujo con el contexto gráfico. |

**Figura 28.4** Los métodos **`Color`** y los métodos relacionados con el color de **`Graphics`**.

Todos los colores se crean a partir de los componentes de rojo, verde y azul. Estos componentes se llaman *valores* RGB (RVA). Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores en punto flotante en el rango de 0.0 a 1.0. La primera parte de RGB define la cantidad de rojo, la segunda define la cantidad de verde y la tercera define la cantidad de azul. El valor RGB más grande será la cantidad de un color en particular. Java permite al programador elegir de entre 256 × 256 × 256 (o aproximadamente 16.7 millones) de colores. Sin embargo, no todas las computadoras son capaces de desplegar todos estos colores. Si éste es el caso, la computadora desplegará el color más cercano posible.

**Error común de programación 28.1**

*Escribir cualquier constante estática de clase de* **`Color`** *con una letra mayúscula inicial, es un error de sintaxis.*

En la figura 28.4 aparecen dos constructores **`Color`**, uno que toma tres argumentos **`int`** y uno que toma tres argumentos **`float`**, en donde cada argumento especifica la cantidad de rojo, verde, y azul, respectivamente. Los valores **`int`** deben estar entre 0 y 255, y los valores **`float`** deben estar entre 0.0 y 1.0. El nuevo objeto **`Color`** contendrá las cantidades especificadas de rojo, verde y azul. Los métodos **`getRed`**, **`getGreen`** y **`getBlue`** de **`Color`** devuelven valores enteros entre 0 y 255 que representan la cantidad de rojo, verde y azul, respectivamente. El método **`setColor`** de **`Graphics`** establece el color de dibujo actual.

La aplicación de la figura 28.5 muestra varios métodos de la figura 28.4 al dibujar rectángulos rellenos y cadenas de diferentes colores.

```
// Figura 28.5: MuestraColores.java
// Demostración de colores
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MuestraColores extends JFrame {
   public MuestraColores()
   {
      super( “Uso de colores” );

      setSize( 400, 130 );
      show();
   } // fin del constructor MuestraColores

   public void paint( Graphics g )
   {
      // establece un nuevo color de dibujo por medio de enteros
      g.setColor( new Color( 255, 0, 0 ) );
      g.fillRect( 25, 25, 100, 20 );
      g.drawString( “RVA actual: “ + g.getColor(), 130, 40 );

      // establece un nuevo color de dibujo por medio de números de punto
         flotante
      g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
      g.fillRect( 25, 50, 100, 20 );
      g.drawString( “RVA actual: “ + g.getColor(), 130, 65 );

      // establece un nuevo color de dibujo por medio de objetos estáticos
         Color
      g.setColor( Color.blue );
      g.fillRect( 25, 75, 100, 20 );
      g.drawString( “RVA actual: “ + g.getColor(), 130, 90 );
```

**Figura 28.5** Muestra cómo establecer y cómo obtener un **`color`**. (Parte 1 de 2.)

```
32
33         // despliega valores individuales RGB
34         Color c = Color.magenta;
35         g.setColor( c );
36         g.fillRect( 25, 100, 100, 20 );
37         g.drawString( "valores RVA: " + c.getRed() + ", " +
38            c.getGreen() + ", " + c.getBlue(), 130, 115 );
39      } // fin del método paint
40
41      public static void main( String args[] )
42      {
43         MuestraColores app = new MuestraColores();
44
45         app.addWindowListener(
46            new WindowAdapter() {
47               public void windowClosing( WindowEvent e )
48               {
49                  System.exit( 0 );
50               } // fin del método windowClosing
51            } // fin de la clase interna anónima
52         ); // fin de addWindowListener
53      } // fin del método main
54   } // fin de la clase MuestraColores
```

**Figura 28.5** Muestra cómo establecer y cómo obtener un **color**. (Parte 2 de 2.)

Cuando comienza la ejecución de la aplicación, se llama al método **paint** de la clase **ShowColors** para pintar la ventana. La línea 19

```
g.setColor( new Color( 255, 0, 0) );
```

utiliza el método **setColor** de **Graphics** para establecer el color actual de dibujo. El método **setColor** recibe un objeto **Color**. La expresión **new Color( 255, 0, 0 )** crea un nuevo objeto **Color** que representa el rojo (valor del rojo **255** y **0** para los colores verde y azul). La línea 20

```
g.fillRect( 25, 25, 100, 20 );
```

utiliza el método **fillRect** de **Graphics** para dibujar un rectángulo relleno con el color actual. Los dos primeros parámetros del método **fillRect** son las coordenadas *x* y *y* de la esquina superior izquierda del rectángulo. El tercer y cuarto parámetros son el ancho y la altura del rectángulo, respectivamente. La línea 21

```
g.drawString( "RVA actual: " + g.getColor(), 130, 40 );
```

utiliza el método **drawString** de **Graphics** para dibujar una cadena (**String**) con el color actual. La expresión **g.getColor()** recibe el color actual desde el objeto **Graphics**. El **Color** devuelto se concatena con la cadena **"RVA actual: "**, lo que resulta en una llamada implícita al método **toString** de la clase **Color**. Observe que la representación de **String** del objeto **Color** contiene el nombre de la clase y el paquete (**java.awt.Color**), y los valores para el rojo, el verde y el azul.

Las líneas 24 a 26 y las líneas 29 a 31 realizan de nuevo las mismas tareas. La línea 24

```
g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
```

utiliza el constructor **Color** con tres argumentos **float** para crear el color verde (**0.0f** para el rojo, **1.0f** para el verde y **0.0f** para el azul). Observe la sintaxis de las constantes. La letra **f** que se agrega a la constan-

te en punto flotante indica que la constante se debe tratar como de tipo **`float`**. Por lo general, las constantes en punto flotante se tratan como de tipo **`double`**.

La línea 29 establece el color de dibujo actual en una de las constantes de **`Color`** predefinidas (**`Color.blue`**). Observe que el nuevo operador no necesita crear una constante. Las constantes de **`Color`** son estáticas, de modo que se definen cuando se carga la clase **`Color`** dentro de memoria en tiempo de ejecución.

La instrucción de las líneas 27 y 38 muestran los métodos **`getRed`**, **`getGreen`** y **`getBlue`** de **`Color`** y el objeto predefinido **`Color.magenta`**.



### Observación de ingeniería de software 28.2

*Para modificar el color, usted debe crear un objeto **`Color`** (o utilizar una de las constantes predefinidas de **`Color`**); no existen métodos **`set`** (establecer) en la clase **`Color`** para modificar las características del color actual.*

Una de las más novedosas características de Java es el componente GUI predefinido ***`JColorChooser`*** (del paquete **`javax.swing`**) para la selección de colores. La aplicación de la figura 28.6 le permite oprimir un botón para desplegar un diálogo **`JColorChooser`**. Cuando selecciona un color y oprime el botón **Aceptar** del diálogo, el color del fondo de la ventana de aplicación cambia.

```
// Figura 28.6: MuestraColores2.java
// Demostración de JColorChooser
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MuestraColores2 extends JFrame {
   private JButton cambiaColor;
   private Color color = Color.lightGray;
   private Container c;

   public MuestraColores2()
   {
      super( “Utilizando JColorChooser” );

      c = getContentPane();
      c.setLayout( new FlowLayout() );

      cambiaColor = new JButton( “Cambia el color” );
      cambiaColor.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               color =
                  JColorChooser.showDialog( MuestraColores2.this,
                     “Elija un color”, color );

               if ( color == null )
                  color = Color.lightGray;

               c.setBackground( color );
               c.repaint();
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( cambiaColor );

      setSize( 400, 130 );
```

**Figura 28.6** Demostración del diálogo **`JColorChooser`**. (Parte 1 de 2.)

```
39            show();
40         } // fin del constructor MuestraColores2
41
42         public static void main( String args[] )
43         {
44            MuestraColores2 app = new MuestraColores2();
45
46            app.addWindowListener(
47               new WindowAdapter() {
48                  public void windowClosing( WindowEvent e )
49                  {
50                     System.exit( 0 );
51                  } // fin del método windowClosing
52               } // fin de la clase interna anónima
53            ); // fin de addWindowListener
54         } // fin de main
55      } // fin de la clase MuestraColores2
```

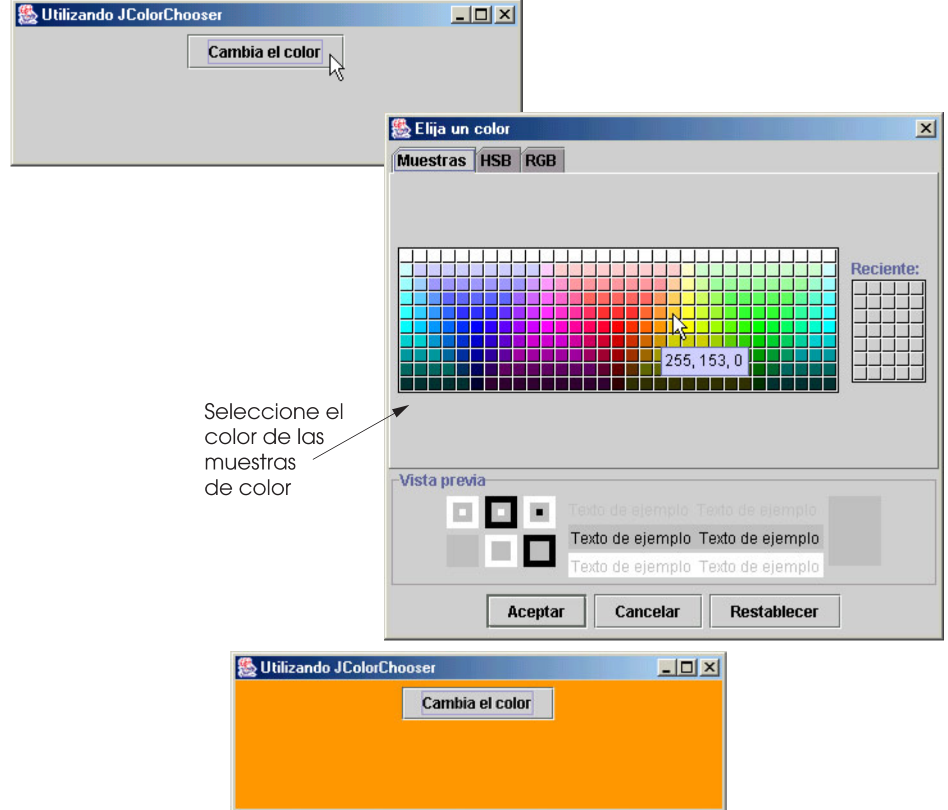



**Figura 28.6** Demostración del diálogo **`JColorChooser`**. (Parte 2 de 2.)

Las líneas 24 a 26 (del método **`actionPerformed`** para **`changeColor`**)

```
color =
   JColorChooser.showDialog( MuestraColores2.this,
      "Elija un color", color );
```

utilizan el método estático ***`showDialog`*** de la clase **`JColorChooser`** para desplegar el diálogo para la elección de colores. Este método devuelve el **`Color`** seleccionado (o **`null`** si el usuario oprime **`Cancelar`** o cierra el diálogo sin presionar **Aceptar**).

El método **showDialog** toma tres argumentos, una referencia al componente (**Compoenent**) padre, un **String** para desplegar en la barra de título del diálogo y el **Color** seleccionado inicialmente para el diálogo. El componente padre es la ventana desde la cual se despliega el diálogo. Mientras el diálogo de elección de color se encuentre en la pantalla, el usuario no puede interactuar con el componente padre. Este tipo de diálogo se llama *diálogo modal* y lo explicaremos en el capítulo 29. Observe la sintaxis especial **ShowColors2.this** que se utiliza en la instrucción anterior. Cuando utiliza una clase interna, usted puede acceder a la referencia **this** del objeto de la clase externa al calificar a **this** con el nombre de la clase externa y el operador punto (**.**).

Una vez que el usuario selecciona el color, las líneas 28 y 29 determinan si **color** es **null**, y si es así, establece **color** al **Color.lightGray** predeterminado. La línea 31

```
c.setBackground( color );
```

utiliza el método **setBackground** para modificar el color del fondo del contenido del panel (representado por **Container c** en este programa). El método **setBackground** es uno de muchos métodos **Component** que pueden utilizarse en la mayoría de los componentes. La línea 32

```
c.repaint();
```

garantiza que el fondo se repinte al llamar a **repaint** para el panel de contenido. Esto programa una llamada al panel de contenido del método **update** del panel, el cual repinta el fondo del panel de contenido con el color de fondo actual.

La segunda captura de pantalla de la figura 28.6 muestra el diálogo predeterminado **JColorChooser** que permite al usuario seleccionar un color de una variedad de *muestras de colores*. Observe que en realidad existen tres fichas a través de la parte superior del diálogo, **Muestras**, **HSB** y **RGB**. Éstas representan tres diferente maneras de seleccionar un color. La ficha **HSB** le permite seleccionar un color basado en *tono*, *saturación* y *brillo*. La ficha **RGB** le permite seleccionar un color mediante el uso de barras de desplazamiento para seleccionar los componentes rojo, verde y azul de un color. Las fichas **HSB** y **RGB** aparecen en la figura 28.7.

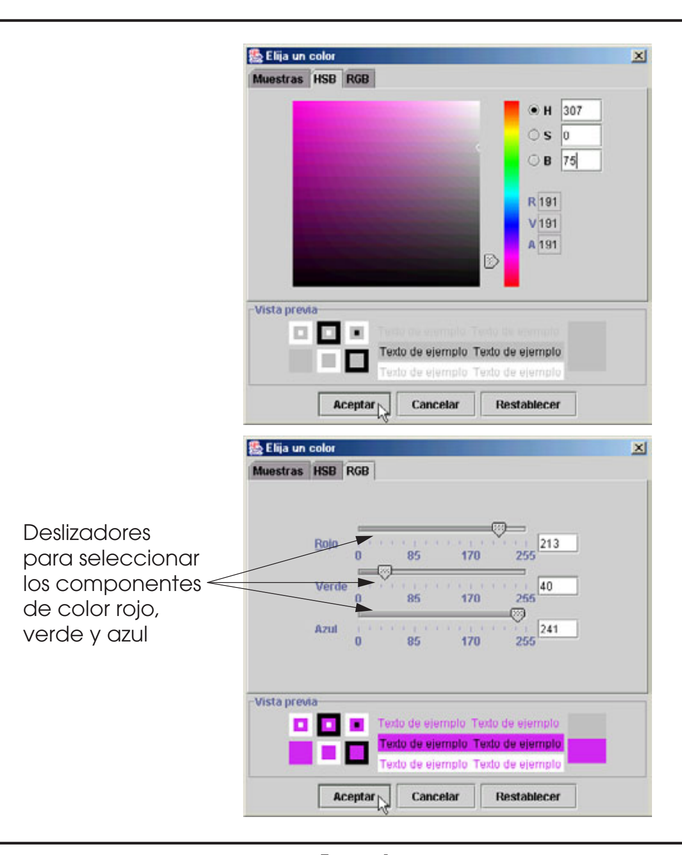



**Figura 28.7** Las fichas **HSB** y **RGB** del diálogo **JColorChooser**.

## 28.4 Control de fuentes

Esta sección presenta los métodos y las constantes para el control de fuentes. La mayoría de los métodos y las constantes de fuentes son parte de la clase **Font**. Algunos métodos de la clase **Font** y de la clase **Graphics** aparecen en la figura 28.8.

El constructor de la clase **Font** toma tres argumentos, el *nombre de la fuente*, el *estilo* y el *tamaño de fuente*. El nombre de la fuente es cualquier fuente soportada por el sistema en donde se ejecuta el programa, tal como las fuentes estándar de Java **Monospaced**, **SansSerif** y **Serif**. El estilo de fuente es **Font.PLAIN**, **Font.ITALIC** o **Font.BOLD** (las constantes estáticas de la clase **Font**). Los estilos de las fuentes pueden utilizarse combinados (por ejemplo, **Font.ITALIC** + **Font.BOLD**). El tamaño de la fuente se mide en puntos. Un *punto* es 1/72 de una pulgada. El método **setFont** de **Graphics** establece la fuente de dibujo actual en su argumento **Fuente**, es decir, la fuente en la que se desplegará el texto.



**Tip de portabilidad 28.2**

*El número de fuentes varía mucho a través de los sistemas. El JDK garantiza que las fuentes **Serif**, **Monospaced**, **SansSerif**, **Dialog** y **DialogInput** estarán disponibles.*

| Método o constante | Descripción |
|---|---|
| `public final static int PLAIN   // clase Font` | Constante que representa un estilo de fuente común. |
| `public final static int BOLD   // clase Font` | Constante que representa un estilo de fuente en negritas. |
| `public final static int ITALIC   // clase Font` | Constante que representa un estilo de fuente en cursivas. |
| `public Font( String nombre, int estilo, int tamaño )` | Crea un objeto **Font** con la fuente, el estilo y el tamaño de la fuente especificada. |
| `public int getStyle ()   // clase Font` | Devuelve un valor entero que indica el estilo de la fuente actual. |
| `public int getSize()   // clase Font` | Devuelve un valor entero que indica el tamaño de la fuente actual. |
| `public String getName()   // clase Font` | Devuelve el nombre de la fuente actual como una cadena. |
| `public String getFamily()   // clase Font` | Devuelve el nombre de la familia de la fuente como una cadena. |
| `public boolean isPlain()   // clase Font` | Verifica si la fuente es de estilo común. Devuelve **true** si la fuente es plana. |
| `public boolean isBold()   // clase Font` | Verifica si la fuente es de estilo negrita. Devuelve **true** si la fuente es negrita. |
| `public boolean isItalic()   // clase Font` | Verifica si la fuente es de estilo cursiva. Devuelve **true** si la fuente es cursiva. |
| `public Font getFont()   // clase Graphics` | Devuelve un objeto **Font** que representa la fuente actual. |
| `public void setFont( Font f)   // clase Graphics` | Establece la fuente actual en la fuente, el estilo y el tamaño determinado por la referencia **f** al objeto **Font**. |

**Figura 28.8** Los métodos de **Font** y las constantes y las fuentes relacionadas con métodos de **Graphics**.

### Error común de programación 28.2

*Especificar una fuente que no está disponible en un sistema, es un error lógico. Java sustituirá a la fuente predeterminada por el sistema.*

El programa de la figura 28.9 despliega texto en cuatro diferentes fuentes con tamaños distintos. El programa utiliza el constructor **`Font`** para inicializar los objetos **`Font`** en las líneas 20, 25, 30 y 37 (cada uno en una llamada al método **`setFont`** de **`Graphics`** para modificar la fuente a dibujar). Cada llamada al constructor **`Font`** pasa un nombre de fuente (Serif, Monospaced o SansSerif) como un **`String`**, un estilo de fuente (**`Font.PLAIN`**, **`Font.ITALIC`** o **`Font.BOLD`**) y el tamaño de la fuente. Una vez que se invoca el método **`setFont`** de **`Graphics`**, todo el texto que se despliegue después de la llamada aparecerá con la nueva fuente hasta que ésta se modifique. Observe que la línea 35 modifica el color del dibujo a rojo, de modo que la siguiente cadena que se despliega aparece en rojo.

### Observación de ingeniería de software 28.3

*Para modificar la fuente, debe crear un nuevo objeto* **`Font`***; no existen métodos establecer (set) en la clase* **`Font`** *para modificar las características de la fuente actual.*

```
1  // Figura 28.9: Fuentes.java
2  // Uso de fuentes
3  import java.awt.*;
4  import javax.swing.*;
5  import java.awt.event.*;
6
7  public class Fuentes extends JFrame {
8     public Fuentes()
9     {
10       super( "Utilizando fuentes" );
11
12       setSize( 420, 125 );
13       show();
14    } // fin del constructor Fuentes
15
16    public void paint( Graphics g )
17    {
18       // establece la fuente actual en Serif (Times), negrita, 12pt
19       // y dibuja una cadena
20       g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
21       g.drawString( "Serif de 12 puntos en negritas.", 20, 50 );
22
23       // establece la fuente actual en Monospaced (Courier),
24       // cursiva, 24pt and draw a string
25       g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
26       g.drawString( "Monospaced de 24 puntos en cursivas.", 20, 70 );
27
28       // establece la fuente actual en SansSerif (Helvetica),
29       // en texto común, 14pt y dibuja una cadena
30       g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
31       g.drawString( "SansSerif de 14 puntos en texto comun.", 20, 90 );
32
33       // establece la fuente actual en Serif (times), negritas/cursivas,
34       // de 18pt y dibuja una cadena
35       g.setColor( Color.red );
36       g.setFont(
37          new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
38       g.drawString( g.getFont().getName() + " " +
```

**Figura 28.9** Uso del método **`setFont`** de **`Graphics`** para modificar las **`Fuentes`**. (Parte 1 de 2.)

```
                         g.getFont().getSize() +
                         " puntos en negritas y cursivas.", 20, 110 );
   } // fin del método paint

   public static void main( String args[] )
   {
      Fuentes app = new Fuentes();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase Fuentes
```

**Figura 28.9** Uso del método **`setFont`** de **`Graphics`** para modificar las **`Fuentes`**. (Parte 2 de 2.)

Con frecuencia es necesario obtener información acerca de la fuente actual, tal como el nombre, el estilo y el tamaño de la fuente. Muchos métodos de **`Font`** utilizados para obtener información de la fuente aparecen en la figura 28.8. El método **`getStyle`** devuelve un valor entero que representa el estilo actual. El valor entero devuelto es **`Font.PLAIN`**, **`Font.ITALIC`**, **`Font.BOLD`** o cualquier combinación de **`Font.PLAIN`**, **`Font.ITALIC`** y **`Font.BOLD`**.

El método **`getSize`** devuelve el tamaño de la fuente en puntos. El método **`getName`** devuelve el nombre de la fuente actual como un **`String`**. El método **`getFamily`** devuelve el nombre de la familia de la fuente a la que pertenece la fuente. El nombre de la familia de la fuente es específica de la plataforma.

### Tip de portabilidad 28.3

*Java utiliza nombres de fuentes estandarizados y los mapea en sistemas específicos de nombres de fuentes para portabilidad. Esto es transparente para el programador.*

Los métodos de **`Font`** también están disponibles para evaluar el estilo de la fuente actual y se resumen en la figura 28.8. El método **`isPlain`** devuelve **`true`** si el estilo de fuente actual es común (plano). El método **`isBold`** devuelve **`true`** si el estilo de la fuente actual es en negritas. El método **`isItalic`** devuelve **`true`** si el estilo de la fuente actual es en cursivas.

En ocasiones, es necesario conocer la información precisa acerca de la métrica de una fuente, tal como la *altura*, el *descendente* (la cantidad de puntos de carácter por debajo de la línea base), el *ascendente* (la cantidad de puntos de carácter por arriba de la línea base) y el *interlineado* (la diferencia entre la altura y el ascendente). La figura 28.10 muestra algunas métricas comunes de las fuentes. Observe que la coordenada pasada a **`drawString`** corresponde a la esquina inferior izquierda de la línea base de la fuente.

La clase ***`FontMetrics`*** define diversos métodos para obtener las características de una fuente. Estos métodos, así como el método ***`getFontMetrics`*** de **`Graphics`**, se encuentran resumidos en la figura 28.11.

El programa de la figura 28.12 utiliza los métodos de la figura 28.11 para obtener información sobre la métrica de dos fuentes.

La línea 19 crea y establece la fuente de dibujo actual en **`SansSerif`** de 12 puntos en negritas. La línea 20 utiliza el método **`getFontMetrics`** de **`Graphics`** para obtener el objeto **`FontMetrics`** para la fuente




**Figura 28.10** Métrica de una fuente.

| Método | Descripción |
|---|---|
| `public int getAscent()` `//clase FontMetrics` | Devuelve un valor que representa el ascendente en puntos de una fuente. |
| `public int getDescent()` `//clase FontMetrics` | Devuelve un valor que representa el descendente en puntos de una fuente. |
| `public int getLeading()` `//clase FontMetrics` | Devuelve un valor que representa el interlineado en puntos de una fuente. |
| `public int getHeight()` `//clase FontMetrics` | Devuelve un valor que representa la altura en puntos de una fuente. |
| `public FontMetrics getFontMetrics()` `//clase Graphics` | Devuelve un valor que representa la altura en puntos de una fuente. |
| `public FontMetrics getFontMetrics( Font f )` `//clase Graphics` | Devuelve el objeto **`FontMetrics`** para el argumento especificado **`Font`**. |

**Figura 28.11** Métodos **`FontMetrics`** y **`Graphics`** para obtener la métrica de una fuente.

```
// Figura 28.12: Metrica.java
// Demostración de los métodos de la clase FontMetrics y
// de la clase Graphics que son útiles para obtener la métrica de una fuente
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Metrica extends JFrame {
   public Metrica()
   {
      super( “Demostrando FontMetrics” );

      setSize( 510, 210 );
      show();
   } // fin del constructor Metrica

   public void paint( Graphics g )
   {
      g.setFont( new Font( “SansSerif”, Font.BOLD, 12 ) );
      FontMetrics fm = g.getFontMetrics();
      g.drawString( “Fuente actual: “ + g.getFont(), 10, 40 );
      g.drawString( “Ascendente: “ + fm.getAscent(), 10, 55 );
      g.drawString( “Descendente: “ + fm.getDescent(), 10, 70 );
```

**Figura 28.12** Cómo obtener información sobre la métrica de una fuente. (Parte 1 de 2.)

```
24        g.drawString( "Altura: " + fm.getHeight(), 10, 85 );
25        g.drawString( "Interlineado: " + fm.getLeading(), 10, 100 );
26
27        Font fuente = new Font( "Serif", Font.ITALIC, 14 );
28        fm = g.getFontMetrics( fuente );
29        g.setFont( fuente );
30        g.drawString( "Fuente actual: " + fuente, 10, 130 );
31        g.drawString( "Ascendente: " + fm.getAscent(), 10, 145 );
32        g.drawString( "Descendente: " + fm.getDescent(), 10, 160 );
33        g.drawString( "Altura: " + fm.getHeight(), 10, 175 );
34        g.drawString( "Interlineado: " + fm.getLeading(), 10, 190 );
35     } // fin del método paint
36
37     public static void main( String args[] )
38     {
39        Metrica app = new Metrica();
40
41        app.addWindowListener(
42           new WindowAdapter() {
43              public void windowClosing( WindowEvent e )
44              {
45                 System.exit( 0 );
46              } // fin del método windowClosing
47           } // fin de la clase interna anónima
48        ); // fin de addWindowListener
49     } // fin de main
50  } // fin de la clase Metrica
```

Demostrando FontMetrics

Fuente actual: java.awt.Font[family=sansserif.bold,name=SansSerif,style=bold,size=12]
Ascendente: 13
Descendente: 3
Altura: 17
Interlineado: 1

*Fuente actual: java.awt.Font[family=serif.italic,name=Serif,style=italic,size=14]*
*Ascendente: 15*
*Descendente: 4*
*Altura: 20*
*Interlineado: 1*

**Figura 28.12** Cómo obtener información sobre la métrica de una fuente. (Parte 2 de 2.)

actual. La línea 21 utiliza una llamada implícita al método **`toString`** de la clase **`Font`** para desplegar la representación de la cadena de la fuente. Las líneas 22 a 25 utilizan los métodos **`FontMetric`** para obtener el ascendente, el descendente, la altura y el interlineado de la fuente.

La línea 27 crea una nueva fuente **`Serif`** de 14 puntos en cursivas. La línea 28 utiliza una segunda versión del método **`getFontMetrics`** de **`Graphics`**, el cual recibe un argumento **`Font`** y devuelve un objeto **`FontMetrics`** correspondiente. Las líneas 31 a 34 obtienen el ascendente, el descendente, la altura y el interlineado para la fuente. Observe que las métricas de la fuente son ligeramente distintas para las dos fuentes.

## 28.5 Cómo dibujar líneas, rectángulos y elipses

Esta sección presenta una variedad de métodos **`Graphics`** para dibujar líneas, rectángulos y elipses. Los métodos y sus parámetros aparecen resumidos en la figura 28.13. Para cada método de dibujo que requiera un parámetro **`ancho`** y **`altura`**, estos valores deben ser positivos. De lo contrario, la figura no se desplegará.

La aplicación de la figura 28.14 muestra el dibujo de una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos redondeados y elipses.

| Método | Descripción |
|---|---|
| **`public void drawLine( int x1, int y1, int x2, int y2 )`** | Dibuja una línea entre el punto **`(x1, y1)`** y el punto **`(x2, y2)`**. |
| **`public void drawRect( int x, int y, int ancho, int altura )`** | Dibuja un rectángulo con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas **`(x, y)`**. |
| **`public void fillRect( int x, int y, int ancho, int altura )`** | Dibuja un rectángulo sólido con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas **`(x, y)`**. |
| **`public void clearRect( int x, int y, int ancho, int altura )`** | Dibuja un rectángulo sólido con el ancho y la altura especificados en el color de fondo actual. La esquina superior izquierda del rectángulo tiene las coordenadas **`(x, y)`**. |
| **`public void drawRoundRect( int x, int y, int ancho, int altura, int anchoArco, int alturaArco )`** | Dibuja un rectángulo con las esquinas redondeadas en el color actual con el ancho y la altura especificados. Los argumentos **`anchoArco`** y **`alturaArco`** determinan el redondeo de las esquinas (vea la figura 28.15). |
| **`public void fillRoundRect( int x, int y, int ancho, int altura, int anchoArco, int alturaArco )`** | Dibuja un rectángulo sólido con las esquinas redondeadas en el color actual con el ancho y la altura especificados. Los argumentos **`anchoArco`** y **`alturaArco`** determinan el redondeo de las esquinas (vea la figura 28.15). |
| **`public void draw3DRect( int x, int y, int ancho, int altura, Boolean b )`** | Dibuja un rectángulo tridimensional en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas **`(x, y)`**. El rectángulo aparece aumentado cuando **`b`** es **`true`** y disminuido cuando **`b`** es **`false`**. |
| **`public void fill3DRect( int x, int y, int ancho, int altura, Boolean b )`** | Dibuja un rectángulo relleno tridimensional en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas **`(x, y)`**. El rectángulo aparece aumentado cuando **`b`** es **`true`** y disminuido cuando **`b`** es **`false`**. |
| **`public void drawOval( int x, int y, int ancho, int altura )`** | Dibuja una elipse en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo delimitador se encuentra en las coordenadas ***`(x, y)`***. La elipse toca los cuatro lados del rectángulo delimitador, en el centro de cada lado (vea la figura 28.16). |
| **`public void fillOval( int x, int y, int ancho, int altura )`** | Dibuja una elipse rellena en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo delimitador se encuentra en las coordenadas ***`(x, y)`***. La elipse toca los cuatro lados del rectángulo delimitador, en el centro de cada lado (vea la figura 28.16). |

**Figura 28.13** Métodos **`Graphics`** que dibujan líneas, rectángulos y elipses.

```
// Figura 28.14: LineasRectangsElips.java
// Dibuja líneas, rectángulos y elipses
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

**Figura 28.14** Demostración del método **`drawLine`** de **`Graphics`**. (Parte 1 de 2.)

```
public class LineasRectangsElips extends JFrame {
   private String s = "Utilizando drawString!";

   public LineasRectangsElips()
   {
      super( "Dibujando lineas, rectangulos y elipses" );

      setSize( 400, 165 );
      show();
   } // fin del constructor LineasRectangsElips

   public void paint( Graphics g )
   {
      g.setColor( Color.red );
      g.drawLine( 5, 30, 350, 30 );

      g.setColor( Color.blue );
      g.drawRect( 5, 40, 90, 55 );
      g.fillRect( 100, 40, 90, 55 );

      g.setColor( Color.cyan );
      g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
      g.drawRoundRect( 290, 40, 90, 55, 20, 20 );

      g.setColor( Color.yellow );
      g.draw3DRect( 5, 100, 90, 55, true );
      g.fill3DRect( 100, 100, 90, 55, false );

      g.setColor( Color.magenta );
      g.drawOval( 195, 100, 90, 55 );
      g.fillOval( 290, 100, 90, 55 );
   } // fin del método paint

   public static void main( String args[] )
   {
      LineasRectangsElips app = new LineasRectangsElips();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase LineasRectangsElips
```

**Figura 28.14** Demostración del método **drawLine** de **Graphics**. (Parte 2 de 2.)

Los métodos **`fillRoundRect`** (línea 28) y **`drawRoundRect`** (línea 29) dibujan rectángulos con esquinas redondeadas. Sus dos primeros argumentos especifican las coordenadas de la esquina superior izquierda del *rectángulo delimitador*, es decir, el área en la que se dibujará el rectángulo. Observe que las coordenadas de la esquina superior izquierda no corresponden al borde del rectángulo redondeado, sino a las coordenadas en donde estaría el borde si el rectángulo tuviera esquinas cuadradas. El tercero y cuarto argumentos especifican el **`ancho`** y la **`altura`** del rectángulo. Sus dos últimos argumentos, **`anchoArco`** y **`alturaArco`**, determinan los diámetros horizontal y vertical de los arcos utilizados para representar las esquinas.

Los métodos **`draw3DRect`** (línea 32) y **`fill3DRect`** (línea 33) toman los mismos argumentos. Los dos primeros argumentos especifican la esquina superior izquierda del rectángulo. Los dos siguientes argumentos especifican el **`ancho`** y la **`altura`** del rectángulo, respectivamente. El último argumento determina si el rectángulo es *aumentado* (**`true`**) o *disminuido* (**`false`**). El efecto tridimensional de **`draw3DRect`** aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de **`fill3DRect`** aparece como dos bordes del rectángulo en el color original de dibujo, y el relleno y los otros dos bordes en un color ligeramente más oscuro. Los rectángulos aumentados tienen el borde superior y el de la izquierda con el color original de dibujo. Los rectángulos disminuidos tienen el borde inferior y el de la derecha con el color original de dibujo. El efecto tridimensional es difícil de apreciar en algunos colores.

La figura 28.15 etiqueta el ancho del arco, la altura del arco, el ancho y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para **`anchoArco`** y **`alturaArco`**, se produce un cuarto de círculo en cada esquina. Cuando **`ancho`**, **`altura`**, **`anchoArco`** y **`alturaArco`** tienen los mismos valores, el resultado es un círculo. Si los valores de **`ancho`** y **`altura`** son los mismos, y los valores de **`anchoArco`** y **`alturaArco`** son 0, el resultado es un cuadrado.

Tanto el método **`drawOval`** como **`fillOvall`** toman los mismos cuatro argumentos. Los dos primeros argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene la elipse. Los dos últimos argumentos especifican el ancho y la altura del rectángulo delimitador, respectivamente. La figura 28.16 muestra una elipse (óvalo) delimitado por un rectángulo. Observe que la elipse toca el centro de los cuatro lados del rectángulo delimitador (el rectángulo delimitador no se despliega en la pantalla).

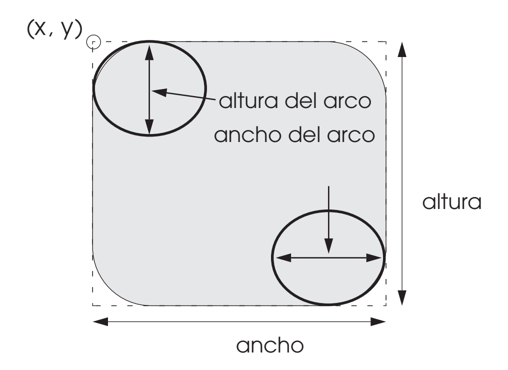



**Figura 28.15** El ancho y la altura del arco para rectángulos redondeados.

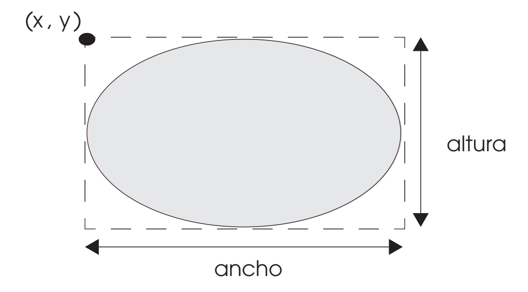



**Figura 28.16** Una elipse limitada por un rectángulo.

## 28.6 Cómo dibujar arcos

Un *arco* es una parte de una elipse. Los ángulos de un arco se miden en grados. Los arcos *barren* desde un *ángulo inicial* el número de grados especificados por al *ángulo del arco*. El ángulo de inicio indica en grados en dónde comienza el arco. El ángulo del arco especifica el número total de grados que el arco barre. La figura 28.17 muestra dos arcos. El conjunto izquierdo de ejes muestra un arco que barre desde cero hasta aproximadamente 110 grados. Los arcos que barren en contra de las manecillas del reloj se miden con *grados positivos*. El conjunto derecho de ejes muestran un arco que barre desde cero hasta aproximadamente 110 grados. Los arcos que barren en la dirección de las manecillas del reloj se miden con *grados negativos*. Observe los cuadros punteados alrededor de los arcos de la figura 28.17. Cuando dibujamos un arco, especificamos un rectángulo delimitador para una elipse. El arco barrerá parte de la elipse. Los métodos ***drawArc*** y ***fillArc*** de **Graphics** para dibujar los arcos aparecen en la figura 28.18.

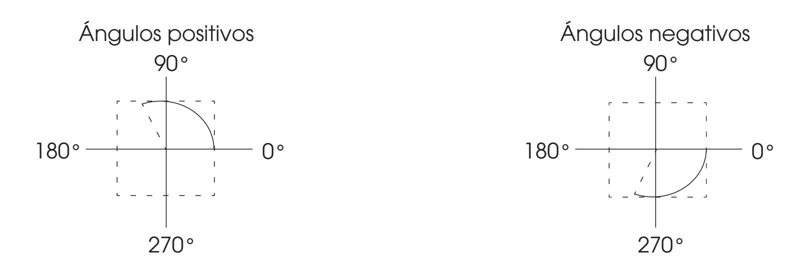



**Figura 28.17** Arcos con ángulos positivos y negativos.

| Método | Descripción |
|---|---|
| **public void drawArc( int x, int y, int ancho, int altura, int anguloInicial, int anguloArco )** | Dibuja un arco relativo a la esquina superior izquierda (**x**, **y**) del rectángulo inscrito con el **ancho** y la **altura** especificados. El segmento de arco se dibuja a partir de **anguloInicial** y barre el número de grados indicado por **anguloArco**. |
| **public void fillArc( int x, int y, int ancho, int altura, int anguloInicial, int anguloArco )** | Dibuja un arco sólido (es decir, un sector) relativo a la esquina superior izquierda **(x, y)** del rectángulo inscrito con el **ancho** y la **altura** especificados. El segmento de arco se dibuja comenzando en **anguloInicial** y barre el número de grados indicado por **anguloArco**. |

**Figura 28.18** Métodos de **Graphics** para dibujar arcos.

El programa de la figura 28.19 muestra los métodos para arcos de la figura 28.18. El programa dibuja seis arcos (tres arcos vacíos y tres arcos rellenos). Para mostrar el rectángulo delimitador que determina en dónde aparece el arco, los tres primeros arcos se despliegan dentro de un rectángulo amarillo que tiene los mismos argumentos **x**, **y**, **ancho** y **altura** como arcos.

```
// Figura 28.19: DibujoArcos.java
// Cómo dibujar arcos
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

**Figura 28.19** Demostración de **drawArc** y **fillArc**. (Parte 1 de 3.)

```
public class DibujoArcos extends JFrame {
   public DibujoArcos()
   {
      super( “Dibujando arcos” );

      setSize( 300, 170 );
      show();
   } // fin del constructor DibujoArcos

   public void paint( Graphics g )
   {
      // comienza en 0 y barre 360 grados
      g.setColor( Color.yellow );
      g.drawRect( 15, 35, 80, 80 );
      g.setColor( Color.black );
      g.drawArc( 15, 35, 80, 80, 0, 360 );

      // comienza en 0 y barre 110 grados
      g.setColor( Color.yellow );
      g.drawRect( 100, 35, 80, 80 );
      g.setColor( Color.black );
      g.drawArc( 100, 35, 80, 80, 0, 110 );

      // comienza en 0 y barre -270 grados
      g.setColor( Color.yellow );
      g.drawRect( 185, 35, 80, 80 );
      g.setColor( Color.black );
      g.drawArc( 185, 35, 80, 80, 0, -270 );

      // comienza en 0 y barre 360 grados
      g.fillArc( 15, 120, 80, 40, 0, 360 );

      // comienza en 270 y barre -90 grados
      g.fillArc( 100, 120, 80, 40, 270, -90 );

      // comienza en 0 y barre -270 grados
      g.fillArc( 185, 120, 80, 40, 0, -270 );
   } // fin del método paint

   public static void main( String args[] )
   {
      DibujoArcos app = new DibujoArcos();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase DibujoArcos
```

**Figura 28.19** Demostración de **drawArc** y **fillArc**. (Parte 2 de 3.)

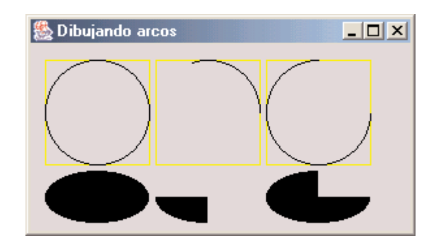



**Figura 28.19** Demostración de **`drawArc`** y **`fillArc`**. (Parte 3 de 3.)

## 28.7 Cómo dibujar polígonos y polilíneas

Los *polígonos* son figuras con múltiples lados. Las *polilíneas* son una serie de puntos conectados. Los métodos gráficos para dibujar polígonos y polilíneas aparecen en la figura 28.19. Observe que algunos métodos requieren un objeto ***`Polygon`*** (del paquete **`java.awt`**). Los constructores de la clase **`Poligono`** también aparecen en la figura 28.20.

| Método | Descripción |
|---|---|
| `public void drawPolygon ( int puntosX[], int puntosY[], int puntos )` | Dibuja un polígono. La coordenada *x* para cada punto se especifica en el arreglo **`puntosX`** y la coordenada *y* de cada punto se especifica en el arreglo **`puntosY`**. El último argumento especifica el número de **`puntos`**. Este método dibuja un polígono cerrado, incluso si el último punto es diferente del primer punto. |
| `public void drawPolyline ( int puntosX[], int puntosY[], int puntos )` | Dibuja una serie de líneas conectadas. La coordenada *x* de cada punto se especifica en el arreglo **`puntosX`** y la coordenada *y* de cada punto se especifica en el arreglo **`puntosY`**. El último argumento especifica el número de **`puntos`**. Si el último punto es diferente del primer punto, la polilínea no se cierra. |
| `public void drawPolygon ( Polygon g )` | Dibuja el polígono cerrado especificado. |
| `public void fillPolygon ( int puntosX[], int puntosY[], int puntos )` | Dibuja un polígono sólido. La coordenada *x* de cada punto se especifica en el arreglo **`puntosX`** y la coordenada *y* de cada punto se especifica en el arreglo **`puntosY`**. El último argumento especifica el número de **`puntos`**. Este método dibuja un polígono cerrado, incluso si el último punto es diferente del primer punto. |
| `public void fillPolygon( Polygon p )` | Dibuja el polígono sólido especificado. El polígono es cerrado. |
| `public Polygon()` | Construye un nuevo objeto polígono. El polígono no contiene punto alguno. |
| `public Polygon ( int valoresX[], int valoresY[], int numeroDePuntos )       // clase Polygon` | Construye un nuevo objeto polígono. El polígono tiene el número de lados que especifica **`numeroDePuntos`**, en donde cada punto consta de una coordenada *x* correspondiente a **`valoresX`**, y una coordenada *y* correspondiente a **`valoresY`**. |

**Figura 28.20** Los métodos de **`Graphics`** para dibujar polígonos, y los constructores de la clase **`Polygon`**

El programa de la figura 28.21 dibuja polígonos y polilíneas por medio de los métodos y constructores de la figura 28.20.

```
// Figura 28.21: DibujoPoligonos.java
// Cómo dibujar polígonos
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DibujoPoligonos extends JFrame {
   public DibujoPoligonos()
   {
      super( "Dibujando poligonos" );

      setSize( 275, 230 );
      show();
   } // fin del constructor DibujoPoligonos

   public void paint( Graphics g )
   {
      int valoresX[] = { 20, 40, 50, 30, 20, 15 };
      int valoresY[] = { 50, 50, 60, 80, 80, 60 };
      Polygon poli1 = new Polygon( valoresX, valoresY, 6 );

      g.drawPolygon( poli1 );

      int valoresX2[] = { 70, 90, 100, 80, 70, 65, 60 };
      int valoresY2[] = { 100, 100, 110, 110, 130, 110, 90 };

      g.drawPolyline( valoresX2, valoresY2, 7 );

      int valoresX3[] = { 120, 140, 150, 190 };
      int valoresY3[] = { 40, 70, 80, 60 };

      g.fillPolygon( valoresX3, valoresY3, 4 );

      Polygon poli2 = new Polygon();
      poli2.addPoint( 165, 135 );
      poli2.addPoint( 175, 150 );
      poli2.addPoint( 270, 200 );
      poli2.addPoint( 200, 220 );
      poli2.addPoint( 130, 180 );

      g.fillPolygon( poli2 );
   } // fin del método paint

   public static void main( String args[] )
   {
      DibujoPoligonos app = new DibujoPoligonos();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
```

**Figura 28.21** Demostración de **drawPolygon** y **fillPolygon**. (Parte 1 de 2.)

```
54            } // fin de la clase interna anónima
55         ); // fin de addWindowListener
56      } // fin de main
57   } // fin de la clase DibujoPoligonos
```

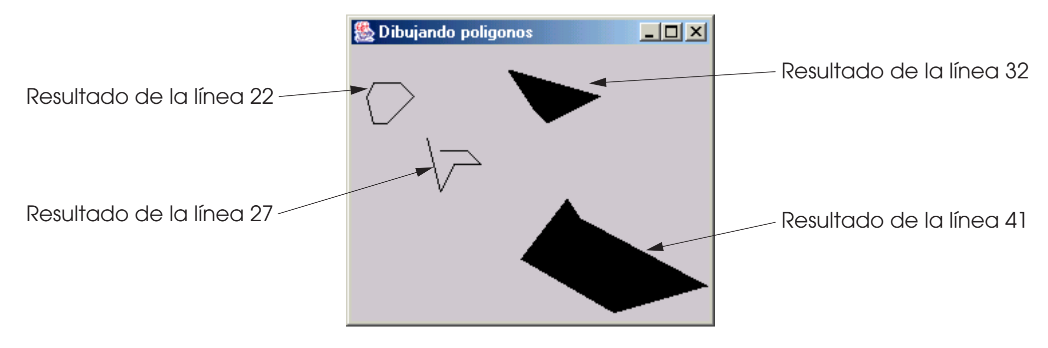



**Figura 28.21** Demostración de **`drawPolygon`** y **`fillPolygon`**. (Parte 2 de 2.)

Las líneas 18 a 20 crean dos arreglos **`int`** y los utilizan para especificar los puntos para **`Polygon poli1`**. La llamada al constructor de **`Polygon`** en la línea 20 recibe el arreglo valores **`valoresX`**, el cual contiene la coordenada *x* de cada punto, el arreglo **`valoresY`**, el cual contiene la coordenada *y* de cada punto, y 6 (el número de puntos en el polígono). La línea 22 despliega **`poli1`**, pasándolo como un argumento del método **`drawPolygon`** de **`Graphics`**.

Las líneas 24 y 25 crean dos arreglos enteros, y los utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo **`valoresX2`** contiene la coordenada *x* de cada punto, y el arreglo **`valoresY2`** contiene la coordenada *y* de cada punto. La línea 27 utiliza el método **`drawPolyline`** de **`Graphics`** para desplegar la serie de líneas conectadas, especificadas con los argumentos **`valoresX2`**, **`valoresY2`** y **`7`** (el número de puntos).

Las líneas 29 y 30 crean dos arreglos enteros, y los utilizan para especificar los puntos de un polígono. El arreglo **`valoresX3`** contiene la coordenada *x* de cada punto, y el arreglo **`valoresY3`** contiene la coordenada *y* de cada punto. La línea 32 despliega un polígono, pasando al método **`fillPoligon`** de **`Graphics`** los dos arreglos (**`valoresX3`** y **`valoresY3`**) y el número de puntos a dibujar (**`4`**).



**Error común de programación 28.3**

*Si el número de puntos especificados en el tercer argumento del método **`drawPolygon`** o del método **`fillPolygon`** es mayor que el número de elementos de los arreglos de coordenadas que definen el polígono a desplegar, se lanza una **`ArrayIndexOutOfBoundsException`**.*

La línea 34 crea **`poli2`** de **`Polygon`** sin puntos. Las líneas 35 a 39 utilizan el método ***`addPoint`*** de **`Polygon`** para agregar pares de coordenadas *x* y *y* al polígono. La línea 41 despliega **`poli2`** de **`Polygon`**, pasándolo al método **`fillPolygon`** de **`Graphics`**.

## 28.8 La API Java2D

El *API Java2D* proporciona capacidades para gráficos de dos dimensiones a programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para el procesamiento de líneas de arte, texto e imágenes de los paquetes **`java.awt`**, **`java.awt.image`**, **`java.awt.color`**, **`java.awt.font`**, **`java.awt.geom`**, **`java.awt.print`** y **`java.awt.image.renderable`**. Las capacidades de la API son demasiado extensas como para cubrirlas en este texto. En esta sección, presentamos una perspectiva general de diversas capacidades de Java2D.

Dibujar con la API Java2D se logra con una instancia de la clase ***`Graphics2D`*** (del paquete **`java.awt`**). La clase **`Graphics2D`** es una subclase de la clase **`Graphics`**, por lo que tiene todas las capacidades gráficas que mostramos anteriormente en este capítulo. De hecho, el objeto real que utilizamos para dibujar en cada método **`paint`** es **`Graphics2D`** que se pasa al método **`paint`**, y se accede a él a través de la referencia de

superclase **g** de **Graphics**. Para acceder a las capacidades de **Graphics2D**, debemos convertir el tipo de la referencia de **Graphics** pasada a **paint** en una referencia **Graphics2D** con una instrucción como

```
Graphics2D g2d = ( Graphics2D ) g;
```

Los programas de las siguientes secciones utilizan esta técnica.

## 28.9 Figuras en Java2D

A continuación, presentamos diversas figuras Java2D del paquete **java.awt.geom**, que incluyen ***Ellipse2D.Double***, ***Rectangle2D.Double***, ***Arc2D.Double***, ***Line2D.Double*** y ***RoundRectangle2D.Double***. Observe la sintaxis del nombre de cada clase. Cada una de estas clases representa una figura con dimensiones especificadas como valores de punto flotante de precisión doble. Existe una versión aparte de cada una, representada con valores de punto flotante de precisión simple (como ***Ellipse2D.Float***). En cada caso, **Double** es una clase interna estática de la clase que se encuentra a la izquierda del operador punto (por ejemplo, **Ellipse2D**). Para utilizar la clase interna estática, simplemente calificamos su nombre con el nombre de la clase externa.

El programa de la figura 28.22 demuestra diversas figuras Java2D y dibuja características como líneas gruesas, rellena figuras con patrones, y dibuja líneas punteadas. Éstas son sólo algunas de las diversas capacidades provistas por Java2D.

```
// Figura 28.22: Figuras.java
// Demostración de algunas figuras de Java2D
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

public class Figuras extends JFrame {
   public Figuras()
   {
      super( "Dibujando figuras de 2D" );

      setSize( 425, 160 );
      show();
   } // fin del constructor Figuras

   public void paint( Graphics g )
   {
      // crea una figura en 2D convirtiendo el tipo de g a Graphics2D
      Graphics2D g2d = ( Graphics2D ) g;

      // dibuja una elipse en 2D rellena con un degradado azul-amarillo
      g2d.setPaint(
         new GradientPaint( 5, 30,           // x1, y1
                            Color.blue,      // Color inicial
                            35, 100,         // x2, y2
                            Color.yellow,    // fin de Color
                            true ) );        // cíclico
      g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );

      // dibuja un rectángulo en 2D en color rojo
      g2d.setPaint( Color.red );
```

**Figura 28.22** Demostración de algunas figuras de Java2D. (Parte 1 de 3.)

```
      g2d.setStroke( new BasicStroke( 10.0f ) );
      g2d.draw(
         new Rectangle2D.Double( 80, 30, 65, 100 ) );

      // dibuja un rectángulo redondeado en 2D con un fondo con
         buferbuffered background
      BufferedImage imagenBuf =
         new BufferedImage(
            10, 10, BufferedImage.TYPE_INT_RGB );

      Graphics2D gg = imagenBuf.createGraphics();
      gg.setColor( Color.yellow ); // dibuja en amarillo
      gg.fillRect( 0, 0, 10, 10 ); // dibuja un rectángulo relleno
      gg.setColor( Color.black );  // dibuja en negro
      gg.drawRect( 1, 1, 6, 6 );   // dibuja un rectángulo
      gg.setColor( Color.blue );   // dibuja en azul
      gg.fillRect( 1, 1, 3, 3 );   // dibuja un rectángulo relleno
      gg.setColor( Color.red );    // dibuja en rojo
      gg.fillRect( 4, 4, 3, 3 );   // dibuja un rectángulo relleno

      // pinta la imagenBuf en el JFrame
      g2d.setPaint(
         new TexturePaint(
            imagenBuf, new Rectangle( 10, 10 ) ) );
      g2d.fill(
         new RoundRectangle2D.Double(
            155, 30, 75, 100, 50, 50 ) );

      // dibuja un arco en 2D en forma de pastel en color blanco
      g2d.setPaint( Color.white );
      g2d.setStroke( new BasicStroke( 6.0f ) );
      g2d.draw(
         new Arc2D.Double(
            240, 30, 75, 100, 0, 270, Arc2D.PIE ) );

      // dibuja líneas en 2D en verde y amarillo
      g2d.setPaint( Color.green );
      g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );

      float guiones[] = { 10 };

      g2d.setPaint( Color.yellow );
      g2d.setStroke(
         new BasicStroke( 4,
                          BasicStroke.CAP_ROUND,
                          BasicStroke.JOIN_ROUND,
                          10, guiones, 0 ) );
      g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
   } // fin del método paint

   public static void main( String args[] )
   {
      Figuras app = new Figuras();

```

**Figura 28.22** Demostración de algunas figuras de Java2D. (Parte 2 de 3.)

```
87         app.addWindowListener(
88            new WindowAdapter() {
89               public void windowClosing( WindowEvent e )
90               {
91                  System.exit( 0 );
92               } // fin del método windowClosing
93            } // fin de la clase interna anónima
94         ); // fin de addWindowListener
95      } // fin de main
96   } // fin de la clase Figuras
```




**Figura 28.22** Demostración de algunas figuras de Java2D. (Parte 3 de 3.)

La línea 21 convierte el tipo de la referencia **Graphics** recibida por **paint** en una referencia **Graphics2D** y la asigna a **g2d** para permitir el acceso a características de Java2D.

La primera figura que dibujamos es una elipse rellena con colores que cambian gradualmente. Las líneas 24 a 29

```
g2d.setPaint(
   new GradientPaint( 5, 30               // x1, y1
                      Color.blue,         // Color inicial
                      35, 100,            // x2, y2
                      Color.yellow,       // fin de Color
                      true ) );           // cíclico
```

invocan al método ***setPaint*** de **Graphics2D** para establecer el objeto ***Paint*** que determina el color de la figura a desplegar. Un objeto **Paint** es un objeto de cualquier clase que implementa la interfaz **java.awt.Paint**. El objeto **Paint** puede ser algo tan sencillo como uno de los objetos **Color** predefinidos que presentamos en la sección 28.3 (la clase **Color** implementa a **Paint**), o el objeto **Paint** puede ser una instancia de las clases ***GradientPaint***, ***SystemColor*** o ***TexturePaint*** de la API Java2D. En este caso, utilizamos un objeto **GradientPaint**.

La clase **GradientPaint** ayuda a dibujar una figura con colores que cambian gradualmente; a lo que se le llama *degradado*. El constructor **GradientPaint** que utilizamos aquí requiere siete argumentos. Los dos primeros argumentos especifican la coordenada inicial del degradado. El tercer argumento especifica el **Color** inicial para el degradado. El cuarto y quinto argumentos especifican la coordenada final del degradado. El sexto argumento especifica el **Color** final del degradado. El último argumento especifica si el degradado es cíclico (**true**) o no cíclico (**false**). Las dos coordenadas determinan la dirección del degradado. La segunda coordenada (*35*, *100*) se encuentra abajo y hacia la derecha de la primera coordenada (*5*, *30*), por lo que el degradado va hacia abajo y hacia la derecha en un ángulo. Este degradado es cíclico (**true**), por lo que el color comienza en azul, poco a poco se vuelve amarillo, y posteriormente regresa poco a poco al azul. Si el degradado no es cíclico, el color sufre una transición del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

La línea 30

```
g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
```

utiliza el método **fill** para dibujar un objeto relleno ***Shape***. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape** (del paquete **java.awt**); en este caso, es una instancia de la clase

**Ellipse2D.Double**. El constructor **Ellipse2D.Double** recibe cuatro argumentos que especifican el rectángulo que limita la elipse a desplegar.

Después, dibujamos un rectángulo rojo con un borde grueso. La línea 33 utiliza **setPaint** para establecer el objeto **Paint** en **Color.red**. La línea 34

```
g2d.setStroke( new BasicStroke( 10.0f ) );
```

utiliza el método ***setStroke*** de **Graphics2D** para establecer las características del borde del rectángulo (o las líneas de cualquier otra figura). El método **setStroke** requiere un objeto ***Stroke*** como su argumento. El objeto **Stroke** es una instancia de cualquier clase que implementa la interfaz **Stroke** (del paquete **java. awt**); en este caso, una instancia de la clase ***BasicStroke***. La clase **BasicStroke** proporciona una variedad de constructores para especificar el ancho de la línea, cómo finaliza la línea (llamada *fin de mayúscula*), cómo unir las líneas (llamado *unión de líneas*) y los atributos para puntear la línea (si se trata de una línea punteada). El constructor aquí especifica que la línea debe tener 10 pixeles de ancho.

Las líneas 35 y 36

```
g2d.draw(
   new Rectangle2D.Double( 80, 30, 65, 100 ) );
```

utilizan el método ***draw*** de **Graphics2D** para dibujar un objeto ***Shape***; en este caso una instancia de la clase **Rectangle2D.Double**. El constructor de **Rectangle2D.Double** recibe cuatro argumentos que especifican la coordenada superior izquierda *x*, la coordenada superior izquierda *y*, el ancho y la altura del rectángulo.

Después dibujamos un rectángulo redondeado relleno con un patrón creado en un objeto ***BufferedImage*** (del paquete **java.awt.image**). Las líneas 39 a 41

```
BufferedImage imagenBuf=
   new BufferedImage(
      10, 10, BufferedImage.TYPE_INT_RGB );
```

crean el objeto **BufferedImage**. La clase **BufferedImage** puede utilizarse para producir imágenes en color y en escala de grises. Este objeto **BufferedImage** es de 10 pixeles de ancho y de 10 pixeles de alto. El tercer argumento del constructor **BufferedImage.TYPE_INT_RGB** indica que la imagen se almacena en color por medio del **esquema de color RGB**.

Para crear el patrón de relleno para el rectángulo redondeado, primero debemos dibujar en el **BufferedImage**. La línea 43

```
Graphics2D gg = imagenBuf.createGraphics();
```

crea un objeto **Graphics2D** que puede utilizarse para dibujar en el **BufferedImage**. Las líneas 44 a 51 utilizan los métodos **setcolor**, **fillRect** y **drawRect** (que explicamos anteriormente en este capítulo) para crear el patrón.

Las líneas 54 a 56

```
g2d.setPaint(
   new TexturePaint(
      imagenBuf, new Rectangle( 10, 10 ) ) );
```

establecen el objeto **Paint** en un nuevo objeto **TexturePaint** (del paquete **java.awt**). Un objeto **TexturePaint** utiliza la imagen almacenada en su **BufferedImage** asociada como la textura de relleno para una figura. El segundo argumento especifica el área del **Rectangulo** del **BufferedImage** que se replicará a través de la textura. En este caso, el **Rectangulo** es del mismo tamaño que **BufferedImage**. Sin embargo, puede utilizarse una parte más pequeña de **BufferedImage**.

Las líneas 57 a 59

```
g2d.fill(
   new RoundRectangle2D.Double(
      155, 30, 75, 100, 50, 50 ) );
```

utilizan el método **fill** de **Graphics2D** para dibujar un objeto **Shape** relleno; en este caso, una instancia de la clase ***RoundRectangle2D.Double***. El constructor **RoundRectangle2D.Double** recibe seis

argumentos que especifican las dimensiones del rectángulo y el ancho y la altura del arco utilizado para determinar el redondeado de las esquinas.

Después dibujamos un arco en forma de pastel con una línea blanca gruesa. La línea 62 establece el objeto **Paint** en **Color.white**. La línea 63 establece el objeto **Stroke** en un nuevo **BasicStroke** para una línea de 6 pixeles de ancho. Las líneas 64 a 66

```
g2d.draw(
   new Arc2D.Double(
      240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
```

utilizan el método **draw** de **Graphics2D** para dibujar un objeto **Shape**; en este caso un **Arc2D.Double**. Los primeros cuatro argumentos del constructor **Arc2D.Double** especifican la coordenada superior izquierda *x*, la coordenada superior izquierda *y*, el ancho y la altura del rectángulo que limita el arco. El quinto argumento especifica el ángulo inicial. El sexto argumento especifica el ángulo del arco. El último argumento especifica cómo se cierra el arco. La constante ***Arc2D.PIE*** indica que el arco se cierra dibujando dos líneas; una a partir del punto de inicio del arco hasta el centro del rectángulo delimitador, y otra desde el centro del rectángulo delimitador hasta el punto final. La clase **Arc2D** proporciona otras dos constantes estáticas para especificar cómo cerrar el arco. La constante ***Arc2D.CHORD*** dibuja una línea desde el punto inicial hasta el punto final. La constante ***Arc2D.OPEN*** especifica que el arco no está cerrado.

Por último, dibujamos dos líneas utilizando objetos ***Line2D***; una continua y otra punteada. La línea 69 establece el objeto **Paint** en **Color.green**. La línea 70

```
g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
```

utiliza el método **draw** de **Graphics2D** para dibujar un objeto **Shape**; en este caso, una instancia de la clase **Line2D.Double**. Los argumentos del constructor **Line2D.Double** especifican las coordenadas iniciales y finales de la línea.

La línea 72 define un arreglo **float** de un elemento que contiene el valor **10**. Este arreglo se utilizará para describir los guiones de la línea punteada. En este caso, cada guión tendrá 10 pixeles de largo. Para crear guiones de diferentes longitudes en un patrón, simplemente proporcione las longitudes de cada uno, como el elemento de un arreglo. La línea 74 establece el objeto **Paint** en **Color.yellow**. Las líneas 75 a 79

```
g2d.setStroke(
   new BasicStroke( 4,
                    BasicStroke.CAP_ROUND,
                    BasicStroke.JOIN_ROUND,
                    10, guiones, 0 ) );
```

establecen el objeto **Stroke** en un nuevo **BasicStroke**. La línea tendrá 4 pixeles de ancho y tendrá bordes redondeados (**BasicStroke.CAP_ROUND**). Si las líneas se unen (como en un rectángulo en las esquinas), la unión de las líneas se redondeará (**BasicStroke.JOIN_ROUND**). El argumento **guiones** especifica las longitudes de los guiones para la línea. El último argumento indica el subíndice de inicio del arreglo guiones para el primer guión del patrón. La línea 80 dibuja una línea con el **Stroke** actual.

Un patrón general es una figura construida a partir de líneas rectas y curvas complejas. Un patrón general se representa con un objeto de la clase ***GeneralPath*** (del paquete **java.awt.geom**). El programa de la figura 28.23 demuestra el dibujo de un patrón general en la figura de una estrella de cinco puntas.

```
// Figura 28.23: Figuras2.java
// Demostración de un patrón general
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.geom.*;

public class Figuras2 extends JFrame {
```

**Figura 28.23** Demostración de **GeneralPaths** de Java2D. (Parte 1 de 3.)

```
   public Figuras2()
   {
      super( “Dibujando figuras en 2D “ );

      setBackground( Color.yellow );
      setSize( 400, 400 );
      show();
   } // fin del constructor Figuras2

   public void paint( Graphics g )
   {
      int puntosX[] =
         { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
      int puntosY[] =
         { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

      Graphics2D g2d = ( Graphics2D ) g;

      // crea una estrella a partir de una serie de puntos
      GeneralPath estrella = new GeneralPath();

      // establece la coordenada inicial del patrón general
      estrella.moveTo( puntosX[ 0 ], puntosY[ 0 ] );

      // crea la estrella--esto no dibuja la estrella
      for ( int k = 1; k < puntosX.length; k++ )
         estrella.lineTo( puntosX[ k ], puntosY[ k ] );

      // cierra la figura
      estrella.closePath();

      // traslada el origen hacia (200, 200)
      g2d.translate( 200, 200 );

      // rota alrededor del origen y dibuja estrellas en colores aleatorios
      for ( int j = 1; j <= 20; j++ ) {
         g2d.rotate( Math.PI / 10.0 );
         g2d.setColor(
            new Color( ( int ) ( Math.random() * 256 ),
                       ( int ) ( Math.random() * 256 ),
                       ( int ) ( Math.random() * 256 ) ) );
         g2d.fill( estrella );   // dibuja una estrella rellena
      } // fin de for
   } // fin del método paint

   public static void main( String args[] )
   {
      Figuras2 app = new Figuras2();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
```

**Figura 28.23** Demostración de **`GeneralPaths`** de Java2D. (Parte 2 de 3.)

```
64            } // fin de la clase interna anónima
65         ); // fin de addWindowListener
66      } // fin de main
67   } // fin de la clase Figuras2
```

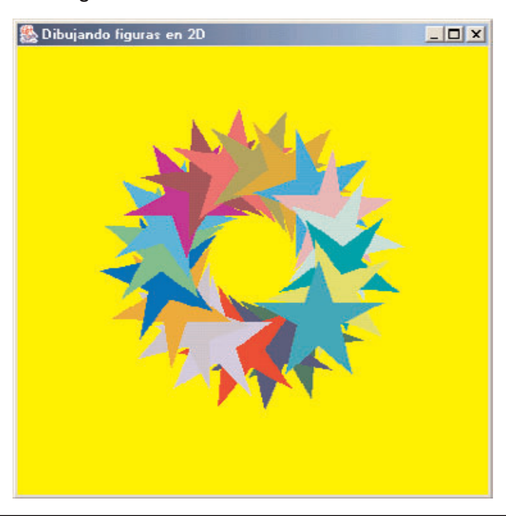



**Figura 28.23** Demostración de **`GeneralPaths`** de Java2D. (Parte 3 de 3.)

Las líneas 20 a 23 definen dos arreglos enteros que representan las coordenadas *x* y *y* de los puntos de la estrella. La línea 28

```
GeneralPath estrella = new GeneralPath();
```

define un objeto **`estrella`** de **`GeneralPath`**.

La línea 31

```
estrella.moveTo( PuntosX[ 0 ], PuntosY[ 0 ] );
```

utiliza el método ***`moveTo`*** de **`GeneralPath`** para especificar el primer punto de la estrella. La estructura **`for`** de las líneas 34 y 35

```
for ( int k = 1; k < PuntosX.length; k++ )
   estrella.lineTo( PuntosX[ k ], PuntosY[ k ] );
```

utilizan el método ***`lineTo`*** de **`GeneralPath`** para dibujar una línea hacia el siguiente punto de la estrella. Cada nueva llamada a **`lineTo`** dibuja una línea desde el punto anterior al punto actual. La línea 38

```
estrella.closePath();
```

utiliza el método ***`closePath`*** de **`GeneralPath`** para dibujar una línea desde el último punto hasta el punto especificado en la última llamada a **`moveTo`**. Esto completa el patrón general.

La línea 41

```
g2d.translate( 200, 200 );
```

utiliza el método ***`translate`*** de **`Graphics2D`** para mover el origen del dibujo hacia la posición (*200*, *200*). Todas las operaciones de dibujo ahora utilizan la posición (*200*, *200*) como (*0*, *0*).

La estructura **`for`** de la línea 44 dibuja 20 veces la **`estrella`**, rotándola alrededor del punto de origen. La línea 45

```
g2d.rotate( Math.PI / 10.0 );
```

utiliza el método ***`rotate`*** de **`Graphics2D`** para rotar la siguiente figura desplegada. El argumento especifica el ángulo de rotación en radianes ( en donde 360° = $2\pi$ radianes). La línea 50 utiliza el método **`fill`** de **`Graphics2D`** para dibujar una versión rellena de la **`estrella`**.

## RESUMEN

- Un sistema de coordenadas es un esquema para identificar cada punto posible en la pantalla.
- La esquina superior izquierda de un componente GUI tiene las coordenadas *(0,0)*. Una par de coordenadas se compone de una coordenada *x* (la coordenada horizontal) y una coordenada *y* (la coordenada vertical).
- Las unidades de coordenadas se miden en pixeles. Un píxel es la unidad de resolución más pequeña de un monitor.
- Un contexto gráfico permite dibujar en la pantalla con Java. Un objeto **Graphics** manipula un contexto gráfico al controlar la manera en que se dibuja la información.
- Los objetos **Graphics** contienen métodos para dibujar, para manipular fuentes, para manipular el color, etcétera.
- Por lo general se llama al método **paint** en respuesta a un *evento* tal como el descubrimiento de una ventana.
- El método **repaint** solicita la llamada al método **update** de **Component** lo más pronto posible para limpiar cualquier dibujo previo en el fondo de **Component**, a continuación **update** llama directamente a **paint**.
- La clase **Color** define métodos y constantes para manipular los colores en un programa en Java.
- Java utiliza los colores RGB, en donde el rojo, el verde y el azul son componentes enteros con un rango entre 0 y 255, o valores de punto flotante con un rango entre 0.0 a 1.0. Mientras más grande sea el valor RGB, mayor será la cantidad de un color en particular.
- Los métodos **getRed**, **getGreen** y **getBlue** de **Color** devuelven valores enteros entre 0 y 255 que representan la cantidad de color rojo, verde y azul en un **Color**.
- La clase **Color** proporciona 13 objetos **Color** predefinidos.
- El método **getColor** de **Graphics** devuelve un objeto **Color** que representa el color de dibujo actual. El método **setColor** de **Graphics** establece el contenido actual del color.
- Java proporciona la clase **JColorChooser** para desplegar un cuadro de diálogo para seleccionar colores.
- El método estático **showDialog** de la clase **JColorChooser** despliega el diálogo para la selección de colores. Este método devuelve el objeto **Color** seleccionado (o **null** si no se seleccionó color alguno).
- El diálogo predeterminado **JColorChooser** le permite seleccionar un color entre una variedad de *muestras de colores*. La ficha **HSB** le permite seleccionar un color basándose en el tono, la saturación y el brillo. La ficha **RGB** le permite seleccionar un color con el uso de barras de desplazamiento para los componentes rojo, verde y azul.
- El método **setBackground** de **Component** (uno de los muchos métodos de **Component** que pueden utilizarse en la mayoría de los componentes de un GUI) modifica el color de fondo de un componente.
- El constructor de la clase **Font** toma tres argumentos, el nombre de la fuente, el *estilo de la fuente* y el *tamaño de la fuente*. El nombre de la fuente corresponde a cualquiera que sea soportada por el sistema. El estilo de la fuente es **Font.PLAIN**, **Font.ITALIC** o **Font.BOLD**. El tamaño de la fuente se mide en puntos.
- El método **setFont** de **Graphics** establece la fuente de dibujo.
- La clase **FontMetrics** define varios métodos para obtener la métrica de la fuente.
- El método **getFontMetrics** de **Graphics** sin argumentos obtiene el objeto **FontMetrics** para la fuente actual. El método **getFontMetrics** de **Graphics** que recibe un argumento **Font**, devuelve el objeto **FontMetrics** correspondiente.
- Los métodos **draw3DRect** y **fill3DRect** toman cinco argumentos que especifican la esquina superior izquierda del rectángulo, el ancho y la altura del rectángulo, y si el rectángulo está aumentado (**true**) o disminuido (**false**).
- Los métodos **drawRoundText** y **fillRoundText** dibujan rectángulos con esquinas redondeadas. Sus dos primeros argumentos especifican la esquina superior izquierda, el tercer y cuarto argumentos especifican el **ancho** y la **altura**, y los dos últimos argumentos, **anchoArco** y **alturaArco**, determinan los diámetros horizontal y vertical de los arcos empleados para representar las esquinas.
- Los métodos **drawOval** y **fillOval** toman los mismos argumentos: la coordenada superior izquierda y el ancho y la altura del rectángulo delimitador que contiene la elipse.
- Un arco es una porción de una elipse. Los arcos barren desde un ángulo inicial hasta el número de grados especificado por el ángulo del arco. El ángulo inicial especifica en dónde comienza el arco y el ángulo del arco especifica el número de grados que barre el arco. Los arcos que barren en contra de las manecillas del reloj se miden en grados positivos y los arcos que se barren en favor de las manecillas del reloj se miden en grados negativos.

- Los métodos **drawArc** y **fillArc** toman los mismos argumentos, la coordenada superior izquierda, el **ancho** y la **altura** del rectángulo delimitador que contiene al arco, y **anguloInicial** y **anguloArco** que definen el barrido del arco.
- Los polígonos son figuras de múltiples lados. Las polilíneas son una serie de puntos conectados.
- Un constructor **Polygon** recibe un arreglo que contiene la coordenada *x* para cada punto, un arreglo que contiene la coordenada *y* para cada punto y el número de puntos del polígono.
- Una versión del método **drawPolygon** de **Graphics** despliega un objeto **Polygon**. Otra versión recibe un arreglo que contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos en el polígono, y despliega el polígono correspondiente.
- El método **drawPolyline** de **Graphics** despliega una serie de líneas conectadas especificada por sus argumentos (un arreglo contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos).
- El método **addPoint** de **Polygon** agrega pares de coordenadas *x* y *y* a un polígono.
- La API Java2D proporciona capacidades para gráficos de dos dimensiones para el procesamiento de líneas de arte, texto e imágenes.
- Para acceder a las capacidades de **Graphics2D**, convierta el tipo de la referencia **Graphics** que se pasa a **paint** en una referencia a **Graphics2D** como en ( **Graphics2D** ) **g**.
- El método **setPaint** de **Graphics2D** establece el objeto **Paint** que determina el color y la textura para la figura a desplegar. Un objeto **Paint** es un objeto de cualquier clase que implementa la interfaz de **java.awt.Paint**. El objeto **Paint** puede ser de un **Color** o una instancia de las clases **GradientPaint**, **SystemColor**, o **TexturePaint** de la API Java2D.
- La clase **GradientPaint** dibuja una figura con un color que cambia gradualmente, llamado *degradado*.
- El método **fill** de **Graphics2D** dibuja un objeto **Shape** relleno. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape**.
- El constructor **Ellipse2D.Double** recibe cuatro argumentos que especifican el rectángulo que delimita la elipse a desplegar.
- El método **setStroke** de **Graphics2D** establece las características de las líneas utilizadas para dibujar la figura. El método **setStroke** requiere un objeto **Stroke** como su argumento. El objeto **Stroke** es una instancia de cualquier clase que implementa la interfaz **Stroke**, tal como **BasicStroke**.
- El método **draw** de **Graphics2D** dibuja un objeto **Shape**. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape**.
- El constructor **Rectangle2D.Double** recibe cuatro argumentos que especifican la coordenada *x* de la esquina superior izquierda, la coordenada *y* de la esquina superior izquierda, el ancho y la altura del rectángulo.
- La clase **BufferedImage** puede utilizarse para producir imágenes en color y en escala de grises.
- Un objeto **TexturePaint** utiliza la imagen almacenada en su objeto **BufferedImage** asociado como la textura de relleno para una figura rellenada.
- El constructor **RoundRectangle2D.Double** recibe seis argumentos que especifican las dimensiones del rectángulo, y el ancho y la altura del arco para determinar las esquinas redondeadas.
- Los cuatro primeros argumentos del constructor **Arc2D.Double** especifican la coordenada *x* de la esquina superior izquierda, la coordenada *y* de la esquina superior izquierda para el arco. El quinto argumento especifica el ángulo inicial. El sexto argumento especifica el ángulo final. El último argumento especifica el tipo del arco (**Arc2D**, **PIE**, **Arc2D**, **CHORD** o **Arc2D**, **OPEN**).
- Los argumentos del constructor **Line2D.Double** especifican las coordenadas inicial y final de la línea.
- Un patrón general es una figura construida a partir de líneas rectas y curvas complejas representadas mediante un objeto de la clase **GeneralPath** (del paquete **java.awt.geom**).
- El método **moveTo** de **GeneralPath** especifica el primer punto del patrón general. El método **lineTo** de **GeneralPath** dibuja una línea hacia el siguiente punto del patrón general. Cada nueva llamada a **lineTo** dibuja una línea desde el punto previo al punto actual. El método **closePath** de **GeneralPath** dibuja una línea desde el último punto hasta el punto especificado en la última llamada a **moveTo**.
- El método **translate** de **Graphics2D** mueve el origen del dibujo hacia una nueva posición. Todas las operaciones de dibujo ahora utilizan la posición como (*0,0*).

## TERMINOLOGÍA

altura del arco
ancho del arco
ángulo
API Java2D
arco limitado por un rectángulo
ascendente
barrido de un arco
clase **Arc2D.Double**
clase **BufferedImage**
clase **Color**
clase **Componente**
clase **Ellipse2D.Double**
clase **Font**
clase **FontMetrics**
clase **GeneralPath**
clase **GradientPaint**
clase **Graphics**
clase **Graphics2D**
clase **Line2D.Double**
clase **Polygon**
clase **Rectangle2D.Double**
clase **RoundRectangle2D.Double**
clase **SystemColor**
clase **TexturePaint**
color de fondo
componente vertical
contexto gráfico
coordenada
coordenada *x*
coordenada *y*
descendente
dibujar un arco
eje *x*
eje *y*
estilo de la fuente
evento
fuente
fuente **Monospaced**
fuente **SansSerif**
fuente **Serif**
grado
grados negativos
grados positivos
interfaz **Paint**
interfaz **Shape**
interfaz **Stroke**
interlineado
línea base
método **addPoint**
método **closePath**
método **draw**
método **draw3DRect**
método **drawArc**
método **drawLine**
método **drawOval**
método **drawPolygon**
método **drawPolyline**
método **drawRect**
método **drawRoundRect**
método **fill**
método **fill3DRect**
método **fillArc**
método **fillOval**
método **fillPolygon**
método **fillRect**
método **fillRoundRect**
método **getAscent**
método **getBlue**
método **getDescent**
método **getFamily**
método **getFont**
método **getFontList**
método **getFontMetrics**
método **getGreen**
método **getHeight**
método **getLeading**
método **getName**
método **getRed**
método **getSize**
método **getStyle**
método **isBold**
método **isItalic**
método **isPlain**
método **lineTo**
método **moveTo**
método **paint**
método **repaint**
método **setColor**
método **setFont**
método **setPaint**
método **setStroke**
método **translate**
método **update**
métrica de la fuente
nombre de la fuente
objeto gráfico
píxel
polígono
polígono cerrado
polígono relleno
proceso controlado por eventos
punto
rectángulo delimitador
sistema de coordenadas
valor RGB

## ERRORES COMUNES DE PROGRAMACIÓN

**28.1** Escribir cualquier constante estática de clase de **Color** con una letra mayúscula inicial, es un error de sintaxis.

**28.2** Especificar una fuente que no está disponible en un sistema, es un error lógico. Java sustituirá a la fuente predeterminada por el sistema.

**28.3** Si el número de puntos especificados en el tercer argumento del método **drawPolygon** o del método **fillPolygon** es mayor que el número de elementos de los arreglos de coordenadas que definen el polígono a desplegar, se lanza una **ArrayIndexOutOfBoundsException**.

## TIPS DE PORTABILIDAD

**28.1** Diferentes pantallas tienen diferentes resoluciones (es decir, varía la densidad de pixeles). Esto puede provocar que los gráficos parezcan de tamaño diferente en diferentes pantallas.

**28.2** El número de fuentes varía mucho a través de los sistemas. El JDK garantiza que las fuentes **Serif**, **Monospaced**, **SansSerif**, **Dialog** y **DialogInput** estarán disponibles.

**28.3** Java utiliza nombres de fuentes estandarizados y los mapea en sistemas específicos de nombres de fuentes para portabilidad. Esto es transparente para el programador.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**28.1** La coordenada superior izquierda (0,0) de una ventana en realidad se encuentra debajo de la barra de título de la ventana. Por esta razón, las coordenadas de dibujo deben ajustarse para dibujar dentro de los bordes de la ventana. La clase **`Container`** (una superclase de todas las ventanas en Java) contiene el método **`getInsets`** que devuelve un objeto **`Instets`** (del paquete **`java.awt`**) para este propósito. Un objeto **`Insets`** contiene cuatro miembros públicos, **`top`**, **`bottom`**, **`left`** y **`right`**, que representan el número de pixeles de cada borde de la ventana hacia el área de dibujo de ésta.

**28.2** Para modificar el color, usted debe crear un objeto **`Color`** (o utilizar una de las constantes predefinidas de **`Color`**); no existen métodos **`set`** (establecer) en la clase **`Color`** para modificar las características del color actual.

**28.3** Para modificar la fuente, debe crear un nuevo objeto **`Font`**; no existen métodos establecer (set) en la clase **`Font`** para modificar las características de la fuente actual.

## EJERCICIOS DE AUTOEVALUACIÓN

**28.1** Complete los espacios en blanco:

a) En Java2D, el método ____________ de la clase ____________ establece las características de una línea que se utiliza para dibujar una línea.
b) La clase ____________ ayuda a definir el relleno para una figura que cambia gradualmente de un color a otro.
c) El método ____________ de la clase **`Graphics`** dibuja líneas entre dos puntos.
d) RGB son las iniciales en inglés para ____________, ____________ y ____________.
e) Los tamaños de las fuentes se miden en unidades llamadas ____________.
f) La clase ____________ ayuda a definir el relleno para una figura que utiliza un patrón dibujado dentro de un objeto de la clase **`BufferedImage`**.

**28.2** Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Los primeros dos argumentos del método **`drawOval`** de **`Graphics`** especifican las coordenadas del centro de la elipse.
b) En el sistema de coordenadas de Java, los valores de *x* se incrementan de izquierda a derecha.
c) El método **`fillPolygon`** dibuja un polígono sólido con el color actual.
d) El método **`drawArc`** permite ángulos negativos.
e) El método **`getSize`** devuelve el tamaño de la fuente actual en centímetros.
f) La coordenada de píxel (*0,0*) se localiza exactamente en el centro del monitor.

**28.3** Encuentre el/los error(es) en cada una de las siguientes instrucciones y explique cómo corregirlos. Asuma que **`g`** es un objeto de **`Graphics`**.

a) **`g.setFont( “SansSerif” );`**
b) **`g.erase( x, y, a, h );   // limpia el rectángulo en (x, y)`**
c) **`Font f = new Font( “Serif”, Font.BOLDITALIC, 12 );`**
d) **`g.setColor ( Color.Yellow );   // cambia el color a amarillo`**

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**28.1** a) **`setStroke`**, **`Graphics2D`**. b) **`GradientPaint`**. c) **`drawLine`**. d) Red, green, blue. e) Puntos. f) **`TexturePaint`**.

**28.2** a) Falso. Los dos primeros argumentos especifican la esquina superior izquierda del rectángulo delimitador.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. Los tamaños de las fuentes se miden en puntos.
f) Falso. La coordenada *(0,0)* corresponde a la esquina superior izquierda de un componente GUI en el cual ocurre el dibujo.

**28.3** a) El método **`setFont`** toma un objeto **`Font`** como argumento, no una cadena.
b) La clase **`Graphics`** no contiene un método **`erase`**. Se debe utilizar el método **`clearRect`**.
c) **`Font.BOLDITALIC`** no es un estilo de fuente válido. Para obtener una fuente en negritas y cursivas, utilice **`Font.BOLD + Font.ITALIC`**.
d) **`Yellow`** de comenzar con una letra minúscula: **`g.setColor( Color.yellow );`**

## EJERCICIOS

**28.4** Complete los espacios en blanco:

a) La clase ____ de la API Java2D se utiliza para definir elipses.

b) Los métodos **`draw`** y **`fill`** de la clase **`Graphics2D`** requieren un objeto de tipo ____________ como argumento.

c) Las tres constantes que especifican el tipo de fuente son ____________, ____________ y ____________.

d) El método ____________ de **`Graphics2D`** establece el color de pintura para las figuras de Java2D.

**28.5** Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) El método **`drawPolygon`** conecta automáticamente los puntos de finalización del polígono.

b) El método **`drawLine`** dibuja una línea entre dos puntos.

c) El método **`fillArc`** utiliza grados para especificar un ángulo.

d) En el sistema de coordenadas de Java, los valores de *y* se incrementan de abajo hacia arriba.

e) La clase **`Graphics`** hereda directamente de la clase **`Object`**.

f) La clase **`Graphics`** es una clase **`abstract`**.

g) La clase **`Font`** hereda directamente desde la clase **`Graphics`**.

**28.6** Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deben estar separados entre sí por 10 pixeles. Utilice el método **`drawOval`** de la clase **`Graphics`**.

**28.7** Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deben estar separados entre sí por 10 pixeles. Utilice el método **`drawArc`**.

**28.8** Modifique su solución del ejercicio 28.6 para dibujar elipses por medio de instancias de la clase **`Ellipse2D.Double`** y del método **`draw`** de la clase **`Graphics2D`**.

**28.9** Escriba un programa que dibuje líneas con longitudes y colores aleatorios.

**28.10** Modifique su solución del ejercicio 28.9 para dibujar líneas aleatorias, con colores y gruesos de línea aleatorios. Utilice la clase **`Line2D.Double`** y el método **`draw`** de la clase **`Graphics2D`** para dibujar las líneas.

**28.11** Escriba un programa que despliegue triángulos generados de manera aleatoria con colores diferentes. Cada triángulo debe rellenarse con un color diferente. Utilice la clase **`GeneralPath`** y el método **`fill`** de la clase **`Graphics2D`** para dibujar los triángulos.

**28.12** Escriba un programa que dibuje de manera aleatoria caracteres de diferentes tamaños y colores.

**28.13** Escriba un programa que dibuje una rejilla de 8 por 8. Utilice el método **`drawLine`**.

**28.14** Modifique su solución del ejercicio 28.13 para dibujar una rejilla por medio de instancias de la clase **`Line2D.Double`** y el método **`draw`** de la clase **`Graphics2D`**.

**28.15** Escriba un programa que dibuje una rejilla de 10 por 10. Utilice el método **`drawRect`**.

**28.16** Modifique su solución al ejercicio 28.15 para dibujar la rejilla por medio de instancias de la clase **`Rectangle2D.Double`** y del método **`draw`** de la clase **`Graphics2D`**.

**28.17** Escriba un programa que dibuje un tetraedro (una pirámide). Utilice la clase **`GeneralPath`** y el método **`draw`** de la clase **`Graphics2D`**.

**28.18** Escriba un programa que dibuje un cubo. Utilice la clase **`GeneralPath`** y el método **`draw`** de la clase **`Graphics2D`**.

**28.19** Escriba una aplicación que simule un protector de pantalla. La aplicación debe dibujar líneas de manera aleatoria con el uso del método **`drawLine`** de la clase **`Graphics`**. Después de dibujar 100 líneas, la aplicación debe limpiarse a sí misma y comenzar a dibujar las líneas de nuevo. Para permitir que el programa dibuje de manera continua, coloque una llamada a **`repaint`** como la última línea del método **`paint`**. ¿Nota usted algún problema con esto en su sistema?

**28.20** Aquí tenemos un poco más. El paquete **`javax.swing`** contiene una clase llamada **`Timer`** que es capaz de llamar al método **`actionPerformed`** de la interfaz **`ActionListener`** en un intervalo fijo de tiempo (especificado en milisegundos). Modifique la solución del ejercicio 28.19 para eliminar la llamada a **`repaint`** desde el método **`paint`**. Defina su clase de modo que implemente **`ActionListener`** (el método **`actionPerformed`** simplemente debe llamar a **`repaint`**). Defina una variable de instancia de tipo **`Timer`** llamada **`crono`** en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
crono = new Timer( 1000, this );
crono.start();
```

Esto crea una instancia de la clase **`Timer`** que llamará al objeto **`actionPerformed`** del objeto **`this`** cada **`1000`** milisegundos (es decir, cada segundo).

**28.21** Modifique su solución del ejercicio 28.20 para permitir al usuario escribir un número de líneas aleatorias que deben dibujarse antes de que la aplicación se limpie a sí misma y comience a dibujar de nuevo las líneas. Utilice **`JTextField`** para obtener el valor. El usuario debe poder escribir un nuevo número dentro de **`JTextField`** en cualquier momento durante la ejecución del programa. [*Nota:* Combinar los componentes Swing del GUI y las guías de dibujo pueden provocar problemas interesantes para los cuales le presentamos soluciones en el capítulo 29.] Por ahora, la primera línea del método **`paint`** debe ser

```
super.paint( g );
```

para garantizar que los componentes GUI se desplieguen apropiadamente. Usted notará que algunas de las líneas dibujadas de manera aleatoria oscurecerán el **`JTextField`**. Utilice una definición interna de la clase para realizar la manipulación de eventos para **`JTextField`**.

**28.22** Modifique su solución al ejercicio 28.20 para elegir de manera aleatoria diferentes figuras a desplegar (utilice los métodos de la clase **`Graphics`**).

**28.23** Modifique su solución del ejercicio 28.22 para utilizar las clases y las capacidades de dibujo de la API Java2D. Para figuras tales como rectángulos y elipses, dibújelas con degradados generados de manera aleatoria (utilice la clase **`GradientPaint`** para generar el degradado).

**28.24** Escriba un programa que utilice el método **`drawPolyline`** para dibujar una espiral.

**28.25** Escriba un programa que introduzca cuatro números y que grafique dichos números en una gráfica de pastel. Utilice la clase **`Arc2D.Double`** y el método **`fill`** de la clase **`Graphics2D`** para realizar el dibujo. Dibuje cada pieza del pastel con un color diferente.

**28.26** Escriba un applet que introduzca cuatro números y que grafique dichos números en una gráfica de barra. Utilice la clase **`Rectangle2D.Double`** y el método **`fill`** de la clase **`Graphics2D`** para realizar el dibujo. Dibuje cada barra con un color diferente.

# 29

# Componentes de la interfaz gráfica de usuario de Java

## Objetivos

- Comprender los principios de diseño de las interfaces gráficas de usuario.
- Crear interfaces gráficas de usuario.
- Comprender los paquetes que contienen componentes de interfaces gráficas de usuario y clases e interfaces para manejo de eventos.
- Crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- Comprender los eventos del ratón y del teclado.
- Comprender y utilizar los administradores de diseño.




*… los profetas más sabios primero se aseguran del evento.*
Horace Walpole

*¿Crees que puedo escuchar estas cosas durante todo el día?*
Lewis Carroll

*Habla en afirmativo; enfatiza tu elección ignorando totalmente todo lo que rechazas.*
Ralph Waldo Emerson

*Paga con tu dinero y toma tus propias decisiones.*
Punch

*Adivina si puedes, elige si te atreves.*
Pierre Corneille

*¡Todo aquel que entra aquí, pierde toda esperanza!*
Dante Alighieri

## Plan general



## 29.1 Introducción

Una *interfaz gráfica de usuario (GUI)* muestra una interfaz ilustrada de un programa. Una GUI proporciona una "apariencia visual" única a un programa. Ya que las GUIs ofrecen a las diversas aplicaciones un conjunto consistente de componentes intuitivos de interfaz de usuario, el usuario no necesita invertir tanto tiempo en recordar qué es lo que hacen las secuencias de teclas y puede utilizar el programa de forma más productiva.

**Observación de apariencia visual 29.1**

*Las interfaces de usuario consistentes permiten a un usuario aprender a utilizar nuevas aplicaciones en menos tiempo.*

Como ejemplo de una GUI, la figura 29.1 contiene una ventana del Internet Explorer con algunos de sus componentes GUI etiquetados. En esta ventana hay una *barra de menús*, la cual contiene menús (**Archivo**, **Edición**, **Ver**, etcétera). Debajo de la barra de menús hay un conjunto de *botones*, cada uno de los cuales tiene una tarea definida en Internet Explorer. Debajo de los botones hay un *campo de texto*, en el que el usuario puede escribir el nombre de un sitio de la World Wide Web que desee visitar. A la izquierda del campo de texto hay una *etiqueta* que indica cuál es el propósito de este campo. Los menús, botones, campos de texto y etiquetas forman parte de la GUI del Internet Explorer. Éstos le permiten interactuar con el programa. En éste y en el siguiente capítulo, mostraremos estos componentes GUI.

Las GUIs se crean a partir de *componentes GUI* (a los que algunas veces se les llama *controles* o "*widgets*": una abreviatura de *accesorios de ventana*). Un componente GUI es un objeto con el que el usuario interactúa mediante el ratón o el teclado. En la figura 29.2 aparecen varios componentes GUI comunes. En las siguientes secciones hablaremos detalladamente sobre cada uno de estos componentes GUI. En el siguiente capítulo hablaremos sobre componentes GUI más avanzados.

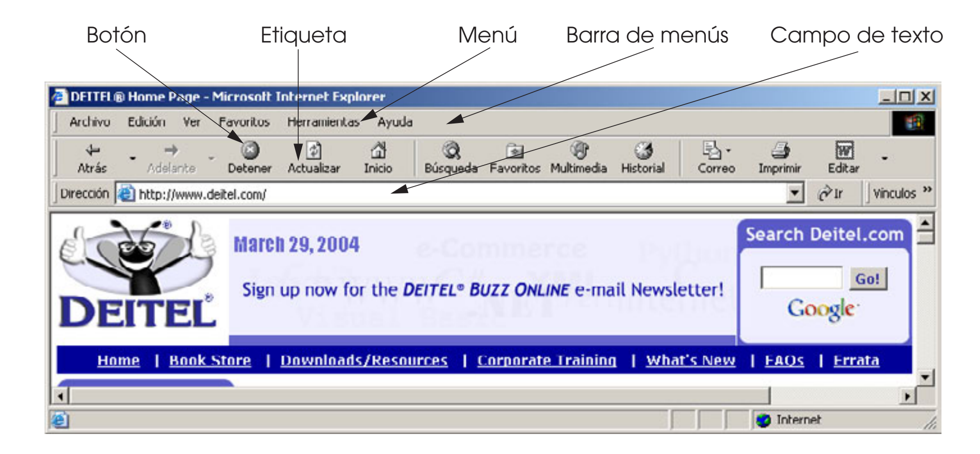



**Figura 29.1** Una ventana de ejemplo del Internet Explorer con los componentes GUI.

| Componente | Descripción |
|---|---|
| `JLabel` | Un área en la que pueden desplegarse iconos o texto que no puede editarse. |
| `JTextField` | Un área en la que el usuario introduce datos a través del teclado. Esta área también puede desplegar información. |
| `JButton` | Un área que desencadena un evento, cuando se hace clic sobre ella. |
| `JCheckBox` | Un componente GUI que puede estar, o no, seleccionado. |
| `JComboBox` | Una lista desplegable de elementos que el usuario puede seleccionar, haciendo clic en un elemento de la lista o escribiendo en el cuadro, si esto está permitido. |
| `JList` | Un área en la que aparece una lista de elementos que el usuario puede seleccionar, haciendo clic una vez en cualquier elemento de la lista. Si hace doble clic en un elemento de la lista, generará un evento de acción. Es posible seleccionar varios elementos. |
| `Jpanel` | Un contenedor en el que pueden colocarse otros componentes. |

**Figura 29.2** Algunos componentes GUI básicos.

## 29.2 Generalidades de Swing

Las clases que se utilizan para crear los componentes GUI de la figura 29.2 forman parte de los *componentes GUI de Swing*, que se encuentran en el paquete **`javax.swing`**. Éstos son los componentes GUI más recientes de la plataforma Java 2. Los *componentes Swing* (como se les llama comúnmente) están escritos, se manipulan y se despliegan completamente en Java (por lo cual se les llama *componentes puros de Java*).

Los componentes GUI originales del paquete *Abstract Windowing Toolkit*, **`java.awt`** (también conocido como *AWT*) están enlazados directamente a las herramientas de la interfaz gráfica de usuario de la plataforma local. Por lo tanto, un programa en Java que se ejecuta en distintas plataformas Java tiene una apariencia distinta, e incluso, algunas veces hasta las interacciones del usuario son distintas en cada plataforma. Juntas, a la apariencia y a la forma en que el usuario interactúa con el programa se les conoce como la *apariencia visual* del programa. Los componentes Swing permiten al programador especificar una apariencia visual distinta para cada plataforma, una apariencia visual uniforme entre todas las plataformas, o incluso puede cambiar la apariencia visual mientras el programa se ejecuta.

### Observación de apariencia visual 29.2

*Los componentes Swing están escritos en Java, por lo que ofrecen un mayor nivel de portabilidad y flexibilidad que los componentes GUI originales de Java del paquete* **`java.awt`**.

Los componentes Swing se conocen comúnmente como *componentes ligeros;* están escritos completamente en Java, por lo que no les afectan las complejas herramientas GUI de la plataforma en la que se utilizan. A los componentes AWT (muchos de los cuales son comparables con los componentes Swing) que se enlazan a la plataforma local se les llama *componentes pesados;* éstos dependen del *sistema de ventanas* de la plataforma local para determinar su funcionalidad y su apariencia visual. Cada componente pesado tiene un *componente asociado* (del paquete ***java.awt.peer***), el cual es responsable de las interacciones entre el componente pesado y la plataforma local para mostrarlo y manipularlo. Varios componentes Swing siguen siendo componentes pesados. En particular, las subclases de ***java.awt.Window*** (como la subclase **JFrame** que utilizamos en capítulos anteriores) que muestran ventanas en la pantalla, aún requieren de una interacción directa con el sistema de ventanas local. Como tales, los componentes GUI pesados de Swing son menos flexibles que muchos de los componentes ligeros que presentaremos.

**Tip de portabilidad 29.1**

*La apariencia de una GUI definida con componentes GUI pesados del paquete* **java.awt** *puede variar entre plataformas. Los componentes pesados se "enlazan" a la GUI de la plataforma "local", la cual varía entre las distintas plataformas.*

La figura 29.3 muestra una jerarquía de herencia de las clases que definen los atributos y comportamientos comunes para la mayoría de los componentes Swing. Cada clase aparece con su nombre y con el nombre completo de su paquete. La mayor parte de la funcionalidad de cada componente GUI se deriva de esas clases. Una clase que hereda de la clase **Component** *es un componente*. Por ejemplo, la clase ***Container*** hereda de la clase **Component**, y ésta hereda de **Object**. Por lo tanto, un objeto **Container** *es un* objeto **Component** y también *es un* **Object**, y un objeto **Component** *es un* **Object**. Una clase que hereda de la clase **Container** *es un* **Container**. Por lo tanto, un objeto **JComponent** *es un* **Container**.



**Observación de ingeniería de software 29.1**

*Para utilizar componentes GUI con efectividad debe comprender las jerarquías de herencia de* ***javax.swing*** *y* ***java.awt****; en especial de las clases* ***Component****,* ***Container*** *y* ***JComponent****, que definen características comunes para la mayoría de los componentes Swing.*

La clase **Component** define los métodos que pueden aplicarse a un objeto de cualquier subclase de **Component**. Dos de los métodos que se originan en la clase **Component** los hemos utilizado con frecuencia hasta este punto del texto: **paint** y **repaint**. Es importante comprender los métodos de la clase **Component**, ya que la mayor parte de la funcionalidad heredada por cada una de las subclases de **Component** está originalmente definida por la clase **Component**. Las operaciones comunes a la mayoría de los componentes GUI (tanto Swing como AWT) se encuentran en la clase **Component**.



**Buena práctica de programación 29.1**

*Estudie los métodos de la clase* ***Component*** *que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de la mayoría de los componentes GUI.*

Un objeto **Container** es una colección de componentes relacionados. En aplicaciones con objetos **JFrame** y en applets, adjuntamos componentes al panel de contenido: un objeto **Container**. La clase **Container** define el conjunto de métodos que pueden aplicarse a un objeto de cualquier subclase de **Container**. Uno de los métodos que se origina en la clase **Container**, y que hemos utilizado frecuentemente hasta este punto del texto, para agregar componentes a un panel de contenido, es **add**. Otro método que se origina en la

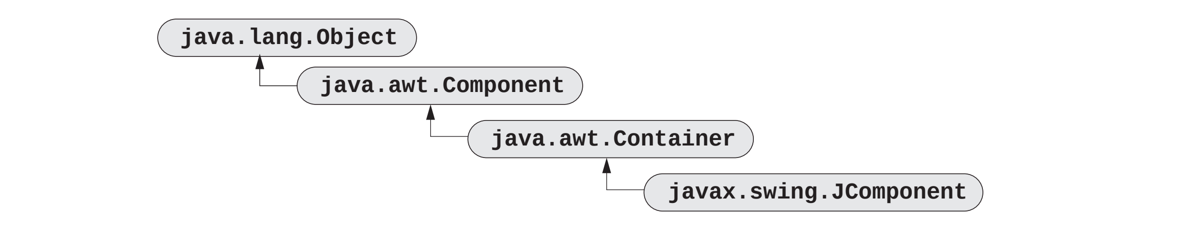



**Figura 29.3** Superclases comunes de muchos de los componentes Swing.

clase **Container** es **setLayout**, el cual hemos utilizado para especificar el administrador de diseño que ayuda a un objeto **Container** a posicionar y ajustar el tamaño de sus componentes.



**Buena práctica de programación 29.2**

*Estudie los métodos de la clase* ***Container*** *que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.*

La clase **JComponent** es la superclase de la mayoría de los componentes Swing. Esta clase define el conjunto de métodos que pueden aplicarse a un objeto de cualquier subclase de **JComponent**.



**Buena práctica de programación 29.3**

*Estudie los métodos de la clase* ***JComponent*** *que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.*

Los componentes Swing que corresponden a subclases de **JComponent** tienen muchas características, las cuales incluyen:

1. Una *apariencia visual adaptable* que puede utilizarse para personalizar la apariencia visual cuando el programa se ejecuta en distintas plataformas.
2. Teclas de acceso directo (llamadas *mnemónicos*) para acceder directamente a los componentes GUI a través del teclado.
3. Herramientas para manejo de eventos comunes, para casos en los que varios componentes GUI inician las mismas acciones en un programa.
4. Breves descripciones del propósito de un componente GUI (que se conocen como *cuadros de información de herramientas*) que aparecen cuando el cursor del ratón se posiciona sobre el componente durante un lapso de tiempo corto.
5. Soporte para tecnologías de asistencia tales como los lectores de pantalla braille para personas ciegas.
6. Soporte para *localización* de la interfaz de usuario; es decir, para personalizar la interfaz de usuario de manera que aparezca en distintos lenguajes y convenciones culturales.

Éstas son sólo algunas de las muchas características de los componentes Swing. En el resto de este capítulo hablaremos sobre varias de estas características.

## 29.3 JLabel

Las etiquetas proporcionan instrucciones de texto o información en una GUI. Éstas se definen mediante la clase ***JLabel***; una subclase de **JComponent**. Una etiqueta muestra una sola línea de *texto de sólo lectura*. Una vez creadas las etiquetas, los programas raras veces cambian el contenido de una etiqueta. La aplicación de la figura 29.4 muestra el uso de **JLabel**.

```
// Figura 29.4: PruebaEtiqueta.java
// Demostración de la clase JLabel.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PruebaEtiqueta extends JFrame {
   private JLabel etiqueta1, etiqueta2, etiqueta3;

   public PruebaEtiqueta()
   {
      super( “Prueba de JLabel” );

      Container c = getContentPane();
```

**Figura 29.4** Demostración de la clase **JLabel**; **PruebaEtiqueta.java**. (Parte 1 de 2.)

```
      c.setLayout( new FlowLayout() );

      // Constructor de JLabel con un argumento tipo cadena
      etiqueta1 = new JLabel( "Etiqueta con texto" );
      etiqueta1.setToolTipText( "Esta es la etiqueta1" );
      c.add( etiqueta1 );

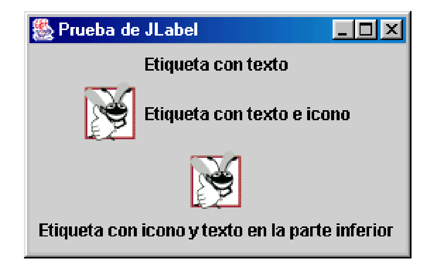
      // Constructor de JLabel con argumentos tipo cadena, Icon
      // y de alineación
      Icon insecto = new ImageIcon( "insecto1.gif" );
      etiqueta2 = new JLabel( "Etiqueta con texto e icono",
                             insecto, SwingConstants.LEFT );
      etiqueta2.setToolTipText( "Esta es la etiqueta2" );
      c.add( etiqueta2 );

      // Constructor de JLabel sin argumentos
      etiqueta3 = new JLabel();
      etiqueta3.setText( "Etiqueta con icono y texto en la parte inferior" );
      etiqueta3.setIcon( insecto );
      etiqueta3.setHorizontalTextPosition(
         SwingConstants.CENTER );
      etiqueta3.setVerticalTextPosition(
         SwingConstants.BOTTOM );
      etiqueta3.setToolTipText( "Esta es la etiqueta3" );
      c.add( etiqueta3 );

      setSize( 275, 170 );
      show();
   } // fin del constructor PruebaEtiqueta

   public static void main( String args[] )
   {
      PruebaEtiqueta ap = new PruebaEtiqueta();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase PruebaEtiqueta
```



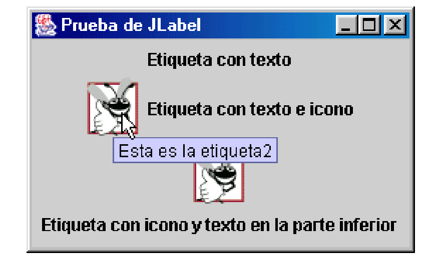




**Figura 29.4** Demostración de la clase **`JLabel`**; **`PruebaEtiqueta.java`**. (Parte 2 de 2.)



### Buena práctica de programación 29.4

*Estudie los métodos de la clase* **`javax.swing.JLabel`** *que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas completas de la clase antes de usarla.*

El programa declara tres referencias **`JLabel`** en la línea 8:

```
private JLabel etiqueta1, etiqueta2, etiqueta3;
```

Los objetos **`Jlabel`** se instancian en el constructor (línea 10). La instrucción

```
etiqueta1 = new JLabel( "Etiqueta con texto" );
```

de la línea 18 crea un objeto **`JLabel`** con el texto **`"Etiqueta con texto"`**. El texto se despliega en la etiqueta. La línea 19

```
etiqueta1.setToolTipText( "Esta es la etiqueta1" );
```

utiliza el método ***`setToolTipText`*** (heredado por la clase **`JLabel`** de la clase **`JComponent`**) para especificar la información de la herramienta (vea la captura de pantalla del lado derecho de la figura 29.4) que aparece cuando el usuario coloca el cursor del ratón sobre la etiqueta de la GUI. Cuando ejecute este programa, intente posicionar el ratón sobre cada una de las etiquetas para ver la información de su herramienta. En la línea 20 se agrega **`etiqueta1`** al panel de contenido.

### Observación de apariencia visual 29.3

*Utilice los cuadros de información de herramienta (establecidos mediante el método* **`setToolTipText`** *de* **`JComponent`***) para agregar texto descriptivo a sus componentes GUI. Este texto ayuda al usuario a determinar el propósito del componente GUI en la interfaz de usuario.*

Muchos componentes Swing pueden mostrar imágenes especificando un objeto **`Icon`** como argumento para su constructor, o utilizando un método que generalmente se llama ***`setIcon`***. Un objeto **`Icon`** es un objeto de cualquier clase que implementa la interfaz ***`Icon`*** (del paquete **`javax.swing`**). Una de esas clases es ***`ImageIcon`*** (del paquete **`javax.swing`**), la cual soporta dos formatos de imagen: *GIF (Formato de intercambio de gráficos)* y *JPEG (Grupo unido de expertos en fotografía)*. Los nombres de archivo para cada uno de estos tipos terminan generalmente con **`.gif`** o **`.jpg`** (o **`.jpeg`**), respectivamente.

En el capítulo 30, hablaremos sobre las imágenes con más detalle. La línea 24:

```
Icon insecto = new ImageIcon( "insecto1.gif" );
```

define un objeto **`ImageIcon`**. El archivo **`insecto1.gif`** contiene la imagen a cargar y a guardar en el objeto **`ImageIcon`**. Este archivo debe encontrarse en el mismo directorio que el programa (en el capítulo 30 veremos cómo puede colocarse el archivo en cualquier otra parte). El objeto **`ImageIcon`** se asigna a la referencia **`Icon`** llamada **`insecto`**. Recuerde que la clase **`ImageIcon`** implementa la interfaz **`Icon`**, por lo tanto, un objeto **`ImageIcon`** *es un* **`Icon`**.

La clase **`JLabel`** soporta el despliegue de objetos **`Icon`**. Las líneas 25 y 26:

```
etiqueta2 = new JLabel( "Etiqueta con texto e icono" );
                        insecto, SwingConstants.LEFT );
```

utilizan otro constructor de **`JLabel`** para crear una etiqueta que muestre el texto **`"Etiqueta con texto e icono"`**, y el objeto **`Icon`** al cual **`insecto`** hace referencia, y se *justifica o alinea hacia la izquierda* (es decir, el icono y el texto se encuentran en la parte izquierda del área de la etiqueta en la pantalla). La interfaz ***`SwingConstants`*** (del paquete **`javax.swing`**) define un conjunto de constantes enteras comunes (como **`SwingConstants.LEFT`**), las cuales se utilizan con muchos componentes Swing. De manera predeterminada, el texto aparece a la derecha de la imagen cuando una etiqueta contiene texto y una imagen. Las alineaciones horizontal y vertical de una etiqueta pueden establecerse mediante los métodos ***`setHorizontalAlignment`*** y ***`setVerticalAlignment`***, respectivamente. La línea 27 especifica el texto de la información de la herramienta para la **`etiqueta2`**. En la línea 28 se agrega **`etiqueta2`** al panel de contenido.



### Error común de programación 29.1

*Olvidar agregar un componente a un contenedor, para que pueda mostrarse en pantalla, es un error lógico en tiempo de ejecución.*

**Error común de programación 29.2**

*Si se agrega a un contenedor un componente que no se haya instanciado, se lanza una excepción* **`NullPointerException`**.

La clase **`JLabel`** proporciona muchos métodos para configurar una etiqueta, después de que se ha instanciado. En la línea 31 se crea un objeto **`JLabel`** y se invoca el constructor sin argumentos (el predeterminado). Dicha etiqueta no tiene texto ni objeto **`Icon`**. La línea 32

```
etiqueta3.setText( "Etiqueta con icono y texto en la parte inferior" );
```

utiliza el método ***`setText`*** de **`JLabel`** para establecer el texto que aparece en la etiqueta. Su método ***`getText`*** correspondiente recupera el texto actual desplegado en una etiqueta. La línea 33

```
etiqueta3.setIcon( insecto );
```

utiliza el método ***`setIcon`*** de **`JLabel`** para establecer el objeto **`Icon`** que aparece en la etiqueta. Su método ***`getIcon`*** correspondiente recupera el objeto **`Icon`** actual desplegado en una etiqueta. Las líneas 34 a 37

```
etiqueta3.setHorizontalTextPosition(
   SwingConstants.CENTER );
etiqueta3.setVerticalTextPosition(
   SwingConstants.BOTTOM );
```

utilizan los métodos ***`setHorizontalTextPosition`*** y ***`setVerticalTextPosition`*** de **`JLabel`** para especificar la posición del texto en la etiqueta. Las instrucciones anteriores indican que el texto se centrará horizontalmente y aparecerá en la parte inferior de la etiqueta. Por lo tanto, el objeto **`Icon`** aparecerá encima del texto. La línea 38 especifica el texto de la información de la herramienta para la **`etiqueta3`**. La línea 39 agrega **`etiqueta3`** al panel de contenido.

## 29.4 Modelo de manejo de eventos

En la sección anterior no hablamos sobre el manejo de eventos, ya que no hay eventos específicos para los objetos **`JLabel`**. Las GUIs están *controladas por eventos* (es decir, generan *eventos* cuando el usuario del programa interactúa con la GUI). Algunas interacciones comunes son mover el ratón, hacer clic con el ratón, hacer clic en un botón, escribir en un campo de texto, seleccionar un elemento de un menú, cerrar una ventana, etcétera. Siempre que ocurre una interacción con el usuario, se envía un evento al programa. La información de los eventos de la GUI se almacena en un objeto de una clase que extiende a **`AWTEvent`**. La figura 29.5 muestra una jerarquía que contiene muchas de las clases de eventos que utilizamos del paquete ***`java.awt.event`***. Muchas de estas clases de eventos las describiremos a lo largo de este capítulo. Los tipos de eventos del paquete

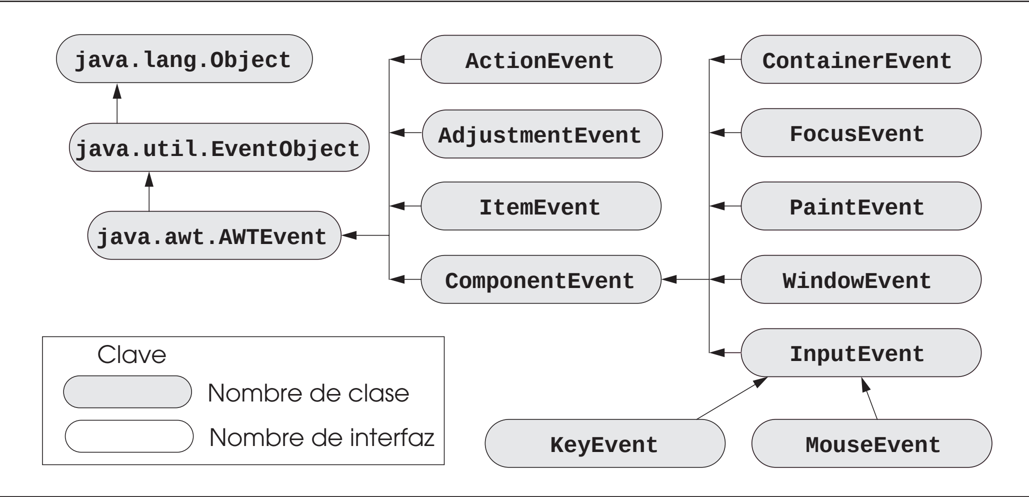



**Figura 29.5** Algunas clases de eventos del paquete **`java.awt.event`**.

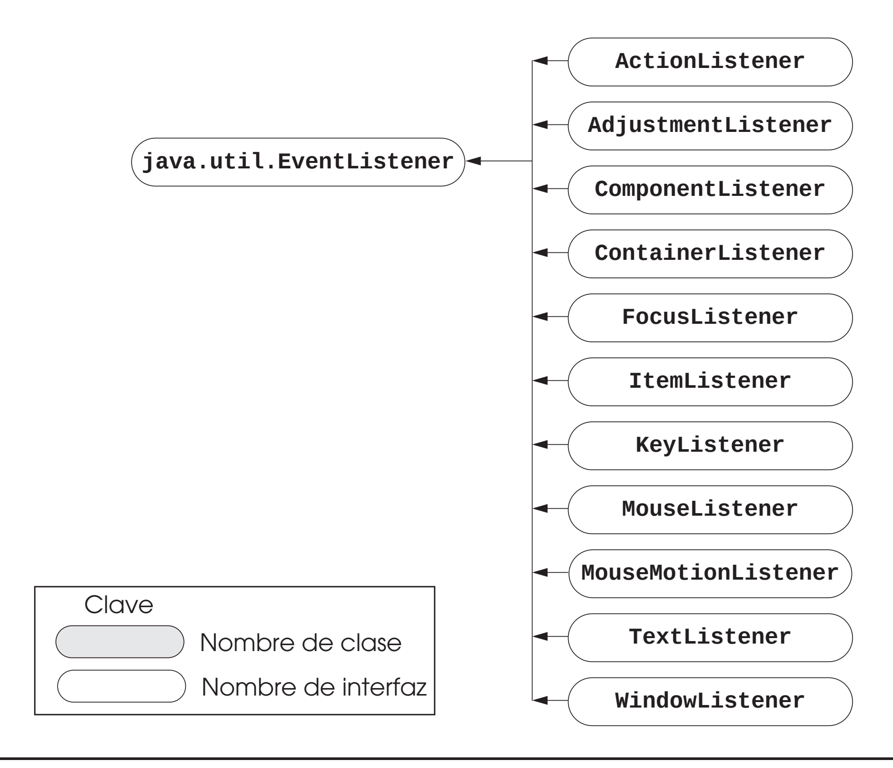



**Figura 29.6** Interfaces que escuchan eventos del paquete **`java.awt.event`**.

***`java.awt.event`*** siguen utilizándose con los componentes Swing. También se han agregado tipos de eventos adicionales, específicos para varios tipos de componentes Swing. Los nuevos tipos de eventos de los componentes Swing están definidos en el paquete ***`javax.swing.event`***.

Para procesar un evento de interfaz gráfica de usuario, el programador debe realizar dos tareas clave: registrar un *componente que escucha eventos* e implementar un *manejador de eventos*. Un componente que escucha un evento GUI es un objeto de una clase que implementa una o más de las interfaces que escuchan eventos correspondientes a los paquetes **`java.awt.event`** y **`javax.swing.event`**. Muchos de los tipos de componentes que escuchan eventos son comunes para los componentes Swing y AWT. Dichos tipos se definen en el paquete **`java.awt.event`** y muchos de ellos aparecen en la figura 29.6 [*Nota:* Un fondo sombreado indica una interfaz en el diagrama]. Los tipos adicionales de componentes que escuchan eventos, específicos de los componentes Swing, se definen en el paquete **`javax.swing.event`**.

Un objeto componente "escucha" tipos específicos de eventos generados en el mismo objeto, o generados por otros objetos (por lo general componentes GUI) en un programa. Un manejador de eventos es un método que se invoca automáticamente en respuesta a un tipo específico de evento. Cada interfaz que escucha eventos especifica uno o más métodos manejadores de eventos que deben definirse en la clase que implementa a la interfaz. Recuerde que las interfaces definen métodos **`abstract`**. Cualquier clase que implemente a una interfaz deberá definir todos los métodos de esa interfaz; en caso contrario, será una clase **`abstract`** y no podrá utilizarse para crear objetos. Al uso de componentes que escuchan eventos en el manejo de eventos se conoce como *modelo de delegación de eventos*; el procesamiento de un evento se delega a un objeto específico en el programa.

Cuando ocurre un evento, el componente GUI que interactuó con el usuario notifica a sus componentes de escucha registrados por medio de una llamada al método manejador de eventos apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime la tecla *Entrar* en un objeto **`JTextField`**, se hace una llamada al método **`actionPerformed`** del componente de escucha registrado. ¿Cómo se registró el manejador de eventos? ¿Cómo es que el componente GUI sabe llamar a **`actionPerformed`** y no a otro método manejador de eventos? Como parte del siguiente ejemplo, responderemos a esas preguntas y mostraremos un diagrama de la interacción.

## 29.5 `JTextField` y `JPasswordField`

Los objetos ***JTextField*** y ***JpasswordField*** (del paquete **`javax.swing`**) son áreas de una sola línea en las que el usuario puede introducir texto a través del teclado, o simplemente puede mostrarse texto. Un objeto **`JPasswordField`** muestra que se escribe un carácter a medida que el usuario introduce los caracteres, pero los oculta debido a que supone que representan una contraseña que sólo debe conocer el usuario. Cuando el usuario escribe datos en un objeto **`JTextField`** o **`JPasswordField`** y oprime la tecla *Entrar*, se produce un evento de acción. Si un componente que escucha eventos está registrado, el evento se procesa y los datos del objeto **`JTextField`** o **`JPasswordField`** pueden utilizarse en el programa. La clase **`JTextField`** extiende a la clase ***JtextComponent*** (del paquete **`javax.swing.text`**), la cual proporciona muchas características comunes a los componentes Swing basados en texto. La clase **`JPasswordField`** extiende a **`JTextField`** y agrega varios métodos específicos para el procesamiento de contraseñas.



**Error común de programación 29.3**

*Utilizar una letra **`f`** minúscula en los nombres de las clases **`JTextField`** o **`JPasswordField`**, es un error de sintaxis.*

La aplicación de la figura 29.7 utiliza las clases **`JTextField`** y **`JPasswordField`** para crear y manipular cuatro campos. Cuando el usuario oprime *Entrar* en el campo que se encuentra activo en ese momento (el componente activo "tiene la atención") se despliega un cuadro de diálogo de mensaje, el cual contiene el texto del campo. Cuando ocurre un evento en el objeto **`JPasswordField`**, la contraseña se revela.

```
// Figura 29.7: PruebaCampoTexto.java
// Demostración de la clase JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PruebaCampoTexto extends JFrame {
   private JTextField texto1, texto2, texto3;
   private JPasswordField contrasenia;

   public PruebaCampoTexto()
   {
      super( “Prueba de JTextField y JPasswordField” );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      // crea el campo de texto con tamaño predeterminado
      texto1 = new JTextField( 10 );
      c.add( texto1 );

      // crea el campo de texto con texto predeterminado
      texto2 = new JTextField( “Escriba el texto aqui” );
      c.add( texto2 );

      // crea el campo de texto con texto predeterminado, con
      // 20 elementos visibles y sin manejador de eventos
      texto3 = new JTextField( “Campo de texto no editable”, 20 );
      texto3.setEditable( false );
      c.add( texto3 );

```

**Figura 29.7** Demostración de **`JTextField`** y **`JPasswordField`**; **`PruebaCampoTexto.java`**. (Parte 1 de 3.)

```
      // crea el campo de contraseña con texto predeterminado
      contrasenia = new JPasswordField( “Texto oculto” );
      c.add( contrasenia );

      ManejadorCampoTexto manejador = new ManejadorCampoTexto();
      texto1.addActionListener( manejador );
      texto2.addActionListener( manejador );
      texto3.addActionListener( manejador );
      contrasenia.addActionListener( manejador );

      setSize( 325, 100 );
      show();
   } // fin del constructor de PruebaCampoTexto

   public static void main( String args[] )
   {
      PruebaCampoTexto ap = new PruebaCampoTexto();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main

   // clase interna privada para el manejo de eventos
   private class ManejadorCampoTexto implements ActionListener {
      public void actionPerformed( ActionEvent evento )
      {
         String cadena = “”;

         if ( evento.getSource() == texto1 )
            cadena = “texto1: “ + evento.getActionCommand();
         else if ( evento.getSource() == texto2 )
            cadena = “texto2: “ + evento.getActionCommand();
         else if ( evento.getSource() == texto3 )
            cadena = “texto3: “ + evento.getActionCommand();
         else if ( evento.getSource() == contrasenia ) {
            JPasswordField contra =
               (JPasswordField) evento.getSource();
            cadena = “contrasenia: “ +
                new String( contra.getPassword() );
         } // fin de else if

         JOptionPane.showMessageDialog( null, cadena );
      } // fin del método actionPerformed
   } // fin de la clase ManejadorCampoTexto
} // fin de la clase PruebaCampoTexto
```

**Figura 29.7** Demostración de **`JTextField`** y **`JPasswordField`**; **`PruebaCampoTexto.java`**. (Parte 2 de 3.)

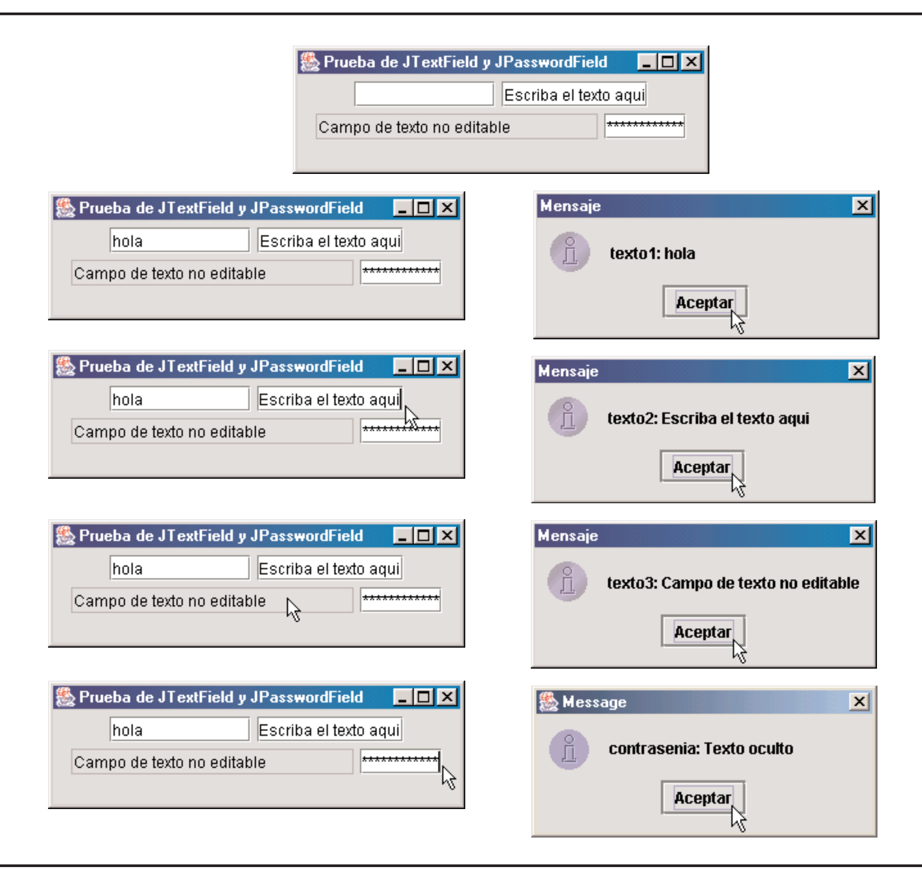



**Figura 29.7** Demostración de **JTextField** y **JPasswordField**; **PruebaCampoTexto.java**. (Parte 3 de 3.)

Las líneas 8 y 9 declaran tres referencias para objetos **JTextField** (**texto1**, **texto2** y **texto3**) y un objeto **JPasswordField** (**contrasenia**). Cada uno de estos objetos son instanciados en el constructor (línea 11). La línea 19

```
texto1 = new JTextField( 10 );
```

define el objeto **texto1** de **JTextField** con **10** columnas de texto. El ancho del campo de texto será el ancho en pixeles del carácter promedio en el tipo de letra actual del campo de texto, multiplicado por **10**. La línea 20 agrega **texto1** al panel de contenido.

La línea 23

```
texto2 = new JTextField( "Escriba el texto aqui" );
```

define el objeto **texto2** de **JTextField** con el texto inicial **"Escriba el texto aqui"** que muestra el campo de texto. El ancho del campo de texto se determina de acuerdo con el texto. La línea 24 agrega **texto2** al panel de contenido.

La línea 28

```
texto3 = new JTextField( "Campo de texto no editable", 20 );
```

define el objeto **texto3** de **JTextField** y hace una llamada al constructor de **JTextField** con dos argumentos: el texto predeterminado **"Campo de texto no editable"** que se muestra en el campo de texto y el número de columnas (**20**). El ancho del campo de texto se determina de acuerdo con el número de columnas especificadas. La línea 29

```
texto3.setEditable( false );
```

utiliza el método ***setEditable*** (heredado en **JTextField** desde la clase **JTextComponent**) para indicar que el usuario no puede modificar el texto del campo de texto. La línea 30 agrega **texto3** al panel de contenido.

La línea 33

```
contrasenia = new JPasswordField( "Texto oculto" );
```

define el objeto **contrasenia** de **JPasswordField** con el texto **"Texto oculto"** a desplegarse en el campo de texto. El ancho del campo de texto se determina de acuerdo con el texto. Observe que el texto aparece como una cadena de asteriscos, cuando se ejecuta el programa. La línea 34 agrega **contrasenia** al panel de contenido.

Para el manejo de eventos de este ejemplo, definimos la clase interna **ManejadorCampoTexto** (líneas 61 a 81). El manejador de la clase **JTextField** (que en breve describiremos detalladamente) implementa la interfaz **ActionListener**. Por lo tanto, toda instancia de la clase **ManejadorCampoTexto** *es un* **ActionListener**. La línea 36

```
ManejadorCampoTexto manejador = new ManejadorCampoTexto();
```

define una instancia de la clase **ManejadorCampoTexto** y la asigna a la referencia **manejador**. Esta instancia se utilizará como el objeto componente que escucha eventos para los objetos **JTextField** y para el objeto **JPasswordField** de este ejemplo.

Las líneas 37 a 40

```
texto1.addActionListener( manejador );
texto2.addActionListener( manejador );
texto3.addActionListener( manejador );
contrasenia.addActionListener( manejador );
```

son las instrucciones de registro de eventos que especifican el objeto componente que escucha eventos para cada uno de los tres objetos **JTextField** y para el objeto **JPasswordField**. Al ejecutarse estas instrucciones, el objeto al que **manejador** hace referencia se queda *escuchando eventos* (es decir, se le notificará cuando ocurra un evento) en estos cuatro objetos. En cada caso, se hace una llamada al método **addActionListener** de la clase **JTextField** para registrar el evento. El método **addActionListener** recibe como su argumento un objeto **ActionListener**. Por lo tanto, podrá proporcionarse cualquier objeto de una clase que implemente la interfaz **ActionListener** (es decir, cualquier objeto que *sea un* **ActionListener**) como argumento de este método. El objeto al que **manejador** hace referencia *es un* **ActionListener**, ya que su clase implementa la interfaz **ActionListener**. Ahora, cuando el usuario oprime *Entrar* en cualquiera de estos cuatro campos, se hace una llamada al método **actionPerformed** (línea 62) de la clase **ManejadorCampoTexto** para manejar el evento.



**Observación de ingeniería de software 29.2**

*El componente que escucha un evento dado deberá implementar la interfaz para escuchar eventos apropiada.*

El método **actionPerformed** utiliza el método **getSource** de su argumento **ActionEvent** para determinar el componente GUI con el que interactuó el usuario, y crea un objeto **String** para mostrarlo en un cuadro de diálogo de mensajes. El método **getActionCommand** de **ActionEvent** devuelve el texto en el objeto **JTextField** que generó el evento. Si el usuario interactúo con el objeto **JPasswordField**, las líneas 73 y 74

```
JPasswordField contra =
   (JPasswordField) evento.getSource();
```

convierten la referencia **Component** devuelta por **evento.getSource()** en una referencia **JPasswordField**, de manera que las líneas 75 y 76

```
cadena = "contrasenia: " +
   new String( contra.getPassword() );
```

puedan utilizar el método ***getPassword*** de **JPasswordField** para obtener la contraseña y crear el objeto **String** a desplegar en pantalla. El método **getPassword** devuelve la contraseña como un arreglo de tipo

**`char`** que se utiliza como argumento para un constructor **`String`**, para crear un objeto **`String`**. La línea 79 muestra un cuadro de mensaje que indica el nombre de la referencia del componente GUI y el texto que escribió el usuario en el campo.

Observe que hasta un objeto **`JTextField`** no editable puede generar un evento. También observe que el texto real de la contraseña aparece cuando oprime *Entrar* en el objeto **`JPasswordField`** (por supuesto, ¡normalmente no haría esto!).



**Error común de programación 29.4**

*Olvidar registrar un objeto manejador de eventos para un tipo de evento de un componente GUI en particular, da como resultado que no se manejen los eventos de ese componente.*

Utilizar una clase separada para definir un componente que escucha eventos es una práctica común de programación para separar la interfaz GUI de la implementación de su manejador de eventos. En el resto de este capítulo, muchos programas utilizan clases separadas de componentes que escuchan eventos para procesar eventos GUI, en un intento por hacer que el código sea más reutilizable. Cualquier clase con el potencial para reutilizarla más allá del ejemplo en el que se introdujo se ha colocado en un paquete, de manera que puede importarse desde otros programas para su reutilización.



**Buena práctica de programación 29.5**

*Utilice clases separadas para procesar eventos GUI.*



**Observación de ingeniería de software 29.3**

*Utilizar clases separadas para manejar eventos GUI produce componentes de software más reutilizables, confiables y legibles, los cuales pueden colocarse en paquetes y utilizarse en muchos programas.*

## 29.5.1 Cómo funciona el manejo de eventos

Ahora veamos cómo funciona el mecanismo de manejo de eventos, utilizando el objeto **`texto1`** de **`JTextField`** del ejemplo anterior. Tenemos dos preguntas de la sección 29.4 que no hemos contestado:

1. ¿Cómo quedó registrado el manejador de eventos?
2. ¿Cómo es que el componente GUI sabe llamar a **`actionPerformed`** y no a otro método manejador de eventos?

Respondemos a la primera pregunta por medio del proceso de registro de eventos que se lleva a cabo en las líneas 37 a 40 del programa. La figura 29.8 muestra un diagrama del objeto **`texto1`** de **`JTextField`** y su manejador de eventos registrado.

Todo objeto **`JComponent`** tiene un objeto de la clase ***`EventListenerList`*** (del paquete **`javax.swing.event`**) llamado ***`listenerList`***, como una variable de instancia. Todos los componentes de es-

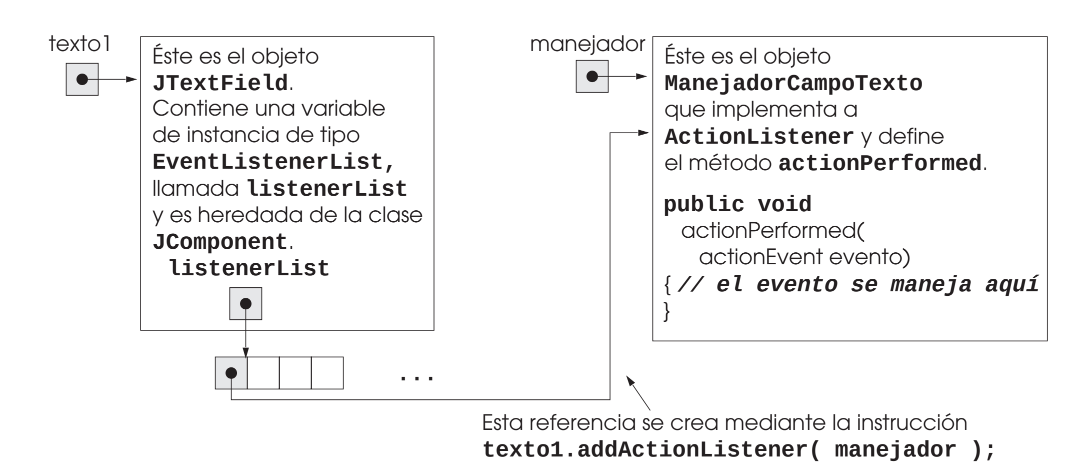



**Figura 29.8** Registro de eventos para el objeto **`texto1`** de **`JTextField`**.

cucha registrados se almacenan en el objeto `listenerList` (diagramado como un arreglo en la figura 29.8). Cuando se ejecuta la instrucción

```
texto1.addActionListener( manejador );
```

de la figura 29.7, se coloca una nueva entrada en el objeto `listenerList` para el objeto `texto1` de `JTextField`, la cual indica la referencia al objeto de escucha y el tipo de componente de escucha (en este caso, `ActionListener`).

El tipo es importante para responder a la segunda pregunta: ¿cómo es que el componente GUI sabe que debe llamar a `actionPerformed`, en vez de a cualquier otro método manejador de eventos? En realidad, todo objeto `JComponent` soporta varios tipos de eventos distintos, incluyendo *eventos del ratón*, *eventos de teclas* y otros más. Cuando ocurre un evento, éste se *despacha* (se envía) solamente a los componentes apropiados que escuchan eventos. El proceso de despachar un evento consiste simplemente en llamar al método manejador de eventos para cada componente de escucha registrado para ese tipo de evento.

Cada tipo de evento tiene su correspondiente interfaz para escuchar eventos. Por ejemplo, los eventos `ActionEvent` son manejados por objetos `ActionListener`, los eventos ***MouseEvent*** son manejados por objetos ***MouseListener*** (y por objetos ***MouseMotionListener***, como veremos más adelante) y los eventos ***KeyEvent*** son manejados por objetos ***KeyListener***. Cuando se genera un evento debido a la interacción del usuario con un componente, éste recibe un *número de identificación (ID) de evento* único, el cual especifica el tipo de evento que ocurrió. El componente GUI utiliza el ID de evento para decidir el tipo de componente de escucha al que debe enviarse el evento, junto con el método al que debe llamarse. En el caso de un evento `ActionEvent`, éste se envía a cada método registrado `actionPerformed` de `ActionListener` (el único método de la interfaz `ActionListener`). En el caso de un `MouseEvent`, éste se envía a todos los objetos `MouseListener` (o `MouseMotionListener`) registrados. El ID del evento `MouseEvent` determina a cuál de los siete distintos métodos manejadores de eventos de ratón se llama. Los componentes GUI manejan toda esta lógica de decisión por usted. Explicaremos otros tipos de eventos e interfaces para escuchar eventos conforme las necesitemos, cuando analicemos cada nuevo componente.

## 29.6 JTextArea

Los objetos ***JTextArea*** proporcionan un área para manipular varias líneas de texto. Al igual que la clase `JTextField`, la clase `JTextArea` hereda de `JTextComponent`, la cual define métodos comunes para objetos `JTextField`, `JTextarea` y varios otros componentes GUI basados en texto.

La aplicación de la figura 29.9 muestra el uso de objetos `JTextArea`. Un objeto `JTextArea` muestra texto que el usuario puede seleccionar. El segundo objeto `JTextArea` no puede editarse, y su propósito es mostrar el texto que el usuario seleccionó en el primer objeto `JTextArea`. Los objetos `JTextArea` no tienen eventos de acción como los objetos `JTextField`. A menudo, un *evento externo* (es decir, un evento generado por otro componente GUI) indica cuándo debe procesarse el texto de un objeto `JTextArea`. Por ejemplo, para enviar un mensaje de correo electrónico, el usuario generalmente hace clic en un botón **Enviar** para tomar el texto del mensaje y enviarlo al destinatario. De manera similar, cuando se edita un documento en un procesador de palabras, usted por lo general guarda el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como…**. En este programa, el botón **Copiar >>>** genera el evento externo que hace que el texto seleccionado del objeto `JTextArea` de la izquierda se copie y se muestre en el objeto `JTextArea` de la derecha.

**Observación de apariencia visual 29.4**

*A menudo, un evento externo determina cuándo debe procesarse el texto de un objeto* ***JTextArea****.*

```
// Figura 29.9: DemoAreaTexto.java
// Cómo copiar texto seleccionado de un área de texto hacia otra.
import java.awt.*;
```

**Figura 29.9** Cómo copiar el texto seleccionado de un área de texto a otra; **DemoAreaTexto.java**. (Parte 1 de 3.)

```
import java.awt.event.*;
import javax.swing.*;

public class DemoAreaTexto extends JFrame {
   private JTextArea t1, t2;
   private JButton copiar;

   public DemoAreaTexto()
   {
      super( “Demostracion de TextArea” );

      Box b = Box.createHorizontalBox();

      String cadena = “Esta es una cadena de demostracion para\n” +
                  “mostrar como copiar texto\n” +
                  “de un objeto TextArea a \n” +
                  “otro objeto TextArea, usando un\n”+
                  “evento externo\n”;

      t1 = new JTextArea( cadena, 10, 15 );
      b.add( new JScrollPane( t1 ) );

      copiar = new JButton( “Copiar >>>” );
      copiar.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               t2.setText( t1.getSelectedText() );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      b.add( copiar );

      t2 = new JTextArea( 10, 15 );
      t2.setEditable( false );
      b.add( new JScrollPane( t2 ) );

      Container c = getContentPane();
      c.add( b );
      setSize( 425, 200 );
      show();
   } // fin del constructor de DemoAreaTexto

   public static void main( String args[] )
   {
      DemoAreaTexto ap = new DemoAreaTexto();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
```

**Figura 29.9** Cómo copiar el texto seleccionado de un área de texto a otra; **`DemoAreaTexto.java`**. (Parte 2 de 3.)

```
58        ); // fin de addWindowListener
59     } // fin de main
60  } // fin de la clase DemoAreaTexto
```

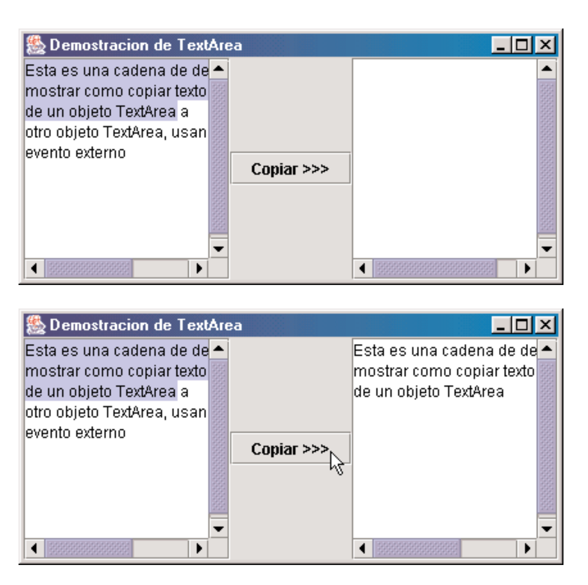



**Figura 29.9** Cómo copiar el texto seleccionado de un área de texto a otra; **`DemoAreaTexto.java`**. (Parte 3 de 3.)

En el método constructor, la línea 15

```
Box b = Box.createHorizontalBox();
```

crea un *contenedor* ***`Box`*** (del paquete **`javax.swing`**) al que se agregarán los componentes GUI. La clase **`Box`** es una subclase de **`java.awt.Container`** que utiliza un administrador de diseño ***`BoxLayout`*** para ordenar los componentes GUI, ya sea en forma horizontal o vertical. En la sección 29.11, hablaremos más sobre los administradores de diseño. La clase **`Box`** proporciona el método estático ***`createHorizontalBox`*** para crear un objeto **`Box`** que ordene automáticamente de izquierda a derecha los componentes que se le agreguen, conforme se vayan agregando.

La aplicación crea instancias de objetos **`JTextArea`** y los asigna a las referencias **`t1`** (línea 23) y **`t2`** (línea 37). Cada objeto **`JTextArea`** tiene **`10`** filas y **`15`** columnas visibles. La línea 23

```
t1 = new JTextArea( cadena, 10, 15 );
```

especifica que la cadena predeterminada **`cadena`** debe mostrarse en el objeto **`JTextArea`**. Un objeto **`JTextArea`** no proporciona barras de desplazamiento, en caso de que haya más texto del que pueda mostrarse en ese objeto. Por esta razón, la línea 24

```
b.add( new JScrollPane( t1 ) );
```

crea un objeto **`JScrollPane`**, el cual se inicializa con el objeto **`t1`** de **`JTextArea`** y con desplazamiento horizontal y vertical, según sea necesario. Después, el objeto **`JScrollPane`** se agrega directamente al contenedor **`Box`** llamado **`b`**.

Las líneas 26 a 35 crean una instancia de un objeto **`JButton`** y la asignan a la referencia **`copiar`** con la etiqueta **`"Copiar >>>"`**; crean una clase interna anónima para manejar el evento **`ActionEvent`** de **`copia`**; y agregan **`copiar`** al objeto contenedor **`Box`** al que **`b`** hace referencia. Este botón proporciona el evento externo que determina cuándo debe copiarse el texto seleccionado de **`t1`** a **`t2`**. Cuando el usuario hace clic en **`Copiar >>>`**, la línea 31

```
t2.setText( t1.getSelectedText() );
```

en **actionPerformed** indica que el método ***getSelectedText*** (heredado a **JTextArea** desde **JTextComponent**) debe regresar el *texto seleccionado* de **t1**. Para seleccionar el texto, arrastre el ratón sobre éste para resaltarlo. Después, el método **setText** cambia el texto en **t2** por el objeto **String** devuelto.

Las líneas 37 a 39 crean el objeto **t2** de **JTextArea** y lo agregan al contenedor **b** de **Box**. Las líneas 41 y 42 obtienen el panel de contenido para la ventana y agregan el objeto **Box** al panel de contenido. El diseño del panel de contenido es administrado por un objeto **BorderLayout**, el cual describiremos en la sección 29.11.2.

[*Nota:* Cuando el texto llega al lado derecho de un objeto **JTextArea**, algunas veces es conveniente que el resto del texto pase a la siguiente línea. A esto se le conoce como *envoltura automática de palabras*.]

**Observación de apariencia visual 29.5**

*Para proporcionar la funcionalidad de envoltura automática de palabras para un objeto **JTextArea**, invoque al método **setLineWrap** con un argumento **true**.*

[*Nota:* Usted puede establecer las *directivas de las barras de desplazamiento* horizontal y vertical para el objeto **JScrollPane** al momento de crearlo, o en cualquier momento mediante los métodos ***setHorizontalScrollBarPolicy*** y ***setVerticalScrollBarPolicy*** de la clase **JScrollPane**.] La clase **JScrollPane** proporciona las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que una barra de desplazamiento debe aparecer siempre; las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que una barra de desplazamiento debe aparecer solamente si es necesario; y las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que una barra de desplazamiento no debe aparecer nunca. Si la directiva de barra de desplazamiento horizontal se establece como **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**, un objeto **JTextArea** adjunto al objeto **JScrollPane** exhibirá un comportamiento de envoltura automática de palabras.

## 29.7 JButton

Un *botón* es un componente en el que el usuario hace clic para desencadenar una acción específica. Un programa en Java puede utilizar varios tipos de botones, que incluyen *botones de comando*, *casillas de verificación*, *botones de conmutación* y *botones de opción*. La figura 29.10 muestra la jerarquía de herencia de los botones Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de ***AbstractButton*** (del paquete **javax.swing**), el cual define muchas de las características comunes para los botones Swing. En esta sección nos concentraremos en los botones que se utilizan generalmente para iniciar un comando. En las siguientes secciones veremos otros tipos de botones.

Un botón de comando genera un evento **ActionEvent** cuando el usuario hace clic con el ratón sobre el botón. Los botones de comando se crean con la clase ***Jbutton***, la cual hereda de la clase **AbstractBut-**

javax.swing.JComponent
javax.swing.AbstractButton
javax.swing.JButton
javax.swing.ToggleButton
javax.swing.JCheckBox
javax.swing.JRadioButton

**Figura 29.10** La jerarquía de botones.

**ton**. Al texto en la cara de un objeto **JButton** se le llama *etiqueta del botón*. Una GUI puede tener muchos objetos **JButton**, pero cada etiqueta de botón debe ser única.

### Observación de apariencia visual 29.6

*Tener más de un objeto **JButton** con la misma etiqueta hace que los objetos **JButton** sean ambiguos para el usuario. Asegúrese de proporcionar una etiqueta única para cada botón.*

La aplicación de la figura 29.11 crea dos objetos **JButton** y muestra que los objetos **JButton** (como los objetos **JLabel**) soportan el despliegue de objetos **Icon**. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la clase interna **ManejadorBoton** (línea 52).

La línea 8 declara dos referencias a instancias de la clase **JButton**: **botonSimple** y **botonElegante**, las cuales se instancian en el constructor (línea 10).

La línea 18

```
botonSimple = new JButton( “Boton simple” );
```

crea **botonSimple** con la etiqueta de botón **“Boton simple”**. La línea 19 agrega el botón al panel de contenido.

Un objeto **JButton** puede mostrar objetos **Icon**. Para proporcionar al usuario un nivel adicional de interactividad visual con la GUI, un objeto **JButton** puede tener también un objeto **Icon** *de sustitución*: un objeto **Icon** que aparece cuando el ratón se posiciona sobre el botón. El icono en el botón cambia a medida que el ratón se aleja y se acerca al área del botón en la pantalla. Las líneas 21 y 22:

```
Icon insecto1 = new ImageIcon( “insecto1.gif” );
Icon insecto2 = new ImageIcon( “insecto2.gif” );
```

crean dos objetos **ImageIcon** que representan al objeto **Icon** predeterminado y al objeto **Icon** de sustitución para el objeto **JButton** creado en la línea 23. Ambas instrucciones asumen que los archivos de imagen están almacenados en el mismo directorio que el programa (por lo general, éste es el caso para las aplicaciones que utilizan imágenes).

```
1  // Figura 29.11: PruebaBoton.java
2  // Creación de objetos JButton.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaBoton extends JFrame {
8     private JButton botonSimple, botonElegante;
9
10    public PruebaBoton()
11    {
12       super( “Prueba de botones” );
13
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       // crea los botones
18       botonSimple = new JButton( “Boton simple” );
19       c.add( botonSimple );
20
21       Icon insecto1 = new ImageIcon( “insecto1.gif” );
22       Icon insecto2 = new ImageIcon( “insecto2.gif” );
23       botonElegante = new JButton( “Boton elegante”, insecto1 );
24       botonElegante.setRolloverIcon( insecto2 );
```

**Figura 29.11** Demostración de botones de comando y de eventos de acción; **PruebaBoton.java**. (Parte 1 de 2.)

```
      c.add( botonElegante );

      // crea una instancia de la clase interna ManejadorBoton
      // para usarla en el manejo de eventos de botón
      ManejadorBoton manejador = new ManejadorBoton();
      botonElegante.addActionListener( manejador );
      botonSimple.addActionListener( manejador );

      setSize( 300, 100 );
      show();
   } // fin del constructor de PruebaBoton

   public static void main( String args[] )
   {
      PruebaBoton ap = new PruebaBoton();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main

   // clase interna para el manejo de eventos de botón
   private class ManejadorBoton implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         JOptionPane.showMessageDialog( null,
            "Usted oprimio: " + e.getActionCommand() );
      } // fin del método actionPerformed
   } // fin de la clase ManejadorBoton
} // fin de la clase PruebaBoton
```

**Figura 29.11** Demostración de botones de comando y de eventos de acción; **`PruebaBoton.java`**. (Parte 2 de 2.)

La línea 23

```
botonElegante = new JButton( "Boton elegante", insecto1 );
```

crea el objeto **`botonElegante`** con el texto predeterminado **`"Boton elegante"`** y el objeto **`insecto1`** de **`Icon`**. De manera predeterminada, el texto se despliega a la derecha del icono. La línea 24

```
botonElegante.setRollOverIcon( insecto2 );
```

utiliza el método ***`setRollOverIcon`*** (la clase **`JButton`** lo hereda de la clase **`AbstractButton`**) para especificar la imagen que se despliega en el botón cuando el usuario coloca el ratón sobre él. La línea 25 agrega el botón al panel de contenido.

### Observación de apariencia visual 29.7

*El uso de iconos de sustitución para objetos **`JButton`** proporciona al usuario una retroalimentación visual, la cual le indica que, si hace clic en el ratón, se realizará la acción del botón.*

Los objetos **`JButton`** (como los **`JTextFields`**) generan **`ActionEvents`**. Como mencionamos anteriormente, un **`ActionEvent`** puede ser procesado por cualquier objeto **`ActionListener`**. Las líneas 29 a 31

```
ManejadorBoton manejador = new ManejadorBoton();
botonElegante.addActionListener( manejador );
botonSimple.addActionListener( manejador );
```

registran un objeto **`ActionListener`** para cada objeto **`JButton`**. La clase interna **`ManejadorBoton`** (líneas 52 a 58) define el método **`actionPerformed`** para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. El método **`getActionCommand`** de **`ActionEvent`** devuelve la etiqueta del botón que generó el evento.

## 29.8 JCheckBox

Los componentes GUI de Swing contienen tres tipos de *botones de estado*: ***`JToggleButton`***, ***`JCheckBox`*** y ***`JRadioButton`***, los cuales tienen valores de tipo encendido/apagado o verdadero/falso. Los **`JToggleButtons`** se utilizan frecuentemente con las *barras de herramientas* (conjuntos de pequeños botones que se encuentran generalmente en una barra, en la parte superior de una ventana). Las clases **`JCheckBox`** y **`JRadioButton`** son subclases de **`JToggleButton`**. Un **`JRadioButton`** es distinto de un **`JCheckBox`** en cuanto a que generalmente hay varios objetos **`JRadioButton`** agrupados, y sólo puede seleccionarse uno de los objetos **`JRadioButton`** (como **`true`**) en cualquier momento dado. En esta sección explicaremos la clase **`JCheckBox`**.

### Observación de apariencia visual 29.8

*La clase **`AbstractButton`** soporta que se despliegue texto e imágenes en un botón, por lo que todas las subclases de **`AbstractButton`** también soportan el despliegue de texto e imágenes.*

La aplicación de la figura 29.12 utiliza dos objetos **`JCheckBox`** para cambiar el estilo de la fuente del texto desplegado en un objeto **`JTextField`**. Un objeto **`JCheckBox`** aplica un estilo de negritas al seleccionarlo, y el otro aplica un estilo de cursivas al seleccionarlo. Si se seleccionan ambos, el estilo de la fuente será en negritas y cursivas. Cuando el programa se ejecuta por primera vez, ninguno de los objetos **`JCheckBox`** está seleccionado (**`true`**).

```
// Figura 29.12: PruebaCasillaVerificacion.java
// Creación de botones JCheckBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

**Figura 29.12** Programa que crea dos botones **`JCheckBox`**; **`PruebaCasillaVerificacion.java`**. (Parte 1 de 3.)

```
public class PruebaCasillaVerificacion extends JFrame {
   private JTextField t;
   private JCheckBox negrita, cursiva;

  public PruebaCasillaVerificacion()
   {
      super( “Prueba de JCheckBox” );

      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      t = new JTextField( “Observe como cambia el estilo de la fuente”, 28 );
      t.setFont( new Font( “TimesRoman”, Font.PLAIN, 14 ) );
      c.add( t );

      // crea los objetos casilla de verificación
      negrita = new JCheckBox( “Negrita” );
      c.add( negrita );

      cursiva = new JCheckBox( “Cursiva” );
      c.add( cursiva );

      ManejadorCasillaVerificacion manejador =
      new ManejadorCasillaVerificacion();
      negrita.addItemListener( manejador );
      cursiva.addItemListener( manejador );

      addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener

      setSize( 325, 100 );
      show();
   } // fin del constructor de PruebaCasillaVerificacion

   public static void main( String args[] )
   {
      new PruebaCasillaVerificacion();
   }

   private class ManejadorCasillaVerificacion implements ItemListener {
   private int valNegrita = Font.PLAIN;
      private int valCursiva = Font.PLAIN;

      public void itemStateChanged( ItemEvent e )
      {
         if ( e.getSource() == negrita )
            if ( e.getStateChange() == ItemEvent.SELECTED )
               valNegrita = Font.BOLD;
```

**Figura 29.12** Programa que crea dos botones **`JCheckBox`**; **`PruebaCasillaVerificacion.java`**. (Parte 2 de 3.)

```
60                 else
61                    valNegrita = Font.PLAIN;
62
63              if ( e.getSource() == cursiva )
64                 if ( e.getStateChange() == ItemEvent.SELECTED )
65                    valCursiva = Font.ITALIC;
66                 else
67                    valCursiva = Font.PLAIN;
68
69              t.setFont(
70                 new Font( “TimesRoman”, valNegrita + valCursiva, 14 ) );
71              t.repaint();
72           } // fin del método itemStateChanged
73        } // fin de la clase interna ManejadorCasillaVerificacion
74     } // fin de la clase PruebaCasillaVerificacion
```

**Figura 29.12** Programa que crea dos botones **JCheckBox**; **PruebaCasillaVerificacion.java**. (Parte 3 de 3.)

Una vez que se crea y se inicializa el objeto **JTextField**, la línea 19

```
t.setFont( new Font( “TimesRoman”, Font.PLAIN, 14 ) );
```

establece en **TimesRoman**, **PLAIN** y en **14** puntos a la fuente del objeto **JTextField**. Después, el constructor crea dos objetos **JCheckBox** mediante las líneas 23 y 26

```
negrita = new JCheckBox( “Negrita” );
cursiva = new JCheckBox( “Cursiva” );
```

El objeto **String** que se pasa al constructor es la *etiqueta de la casilla de verificación* que aparece a la derecha del objeto **JCheckBox**, de manera predeterminada.

Cuando el usuario hace clic en un objeto **JCheckBox** se genera un **ItemEvent**, el cual puede ser manejado por un **ItemListener** (cualquier objeto de una clase que implemente la interfaz **ItemListener**). Un objeto **ItemListener** debe definir el método ***itemStateChanged***. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la clase interna **ManejadorCasillaVerificacion** (líneas 51 a 73). Las líneas 29 a 31

```
ManejadorCasillaVerificacion manejador = new ManejadorCasillaVerificacion();
negrita.addItemListener( manejador );
cursiva.addItemListener( manejador );
```

crean una instancia de la clase **ManejadorCasillaVerificacion** y se registra con el método ***addItemListener*** como el objeto **ItemListener** para los objetos **JCheckBox** **negrita** y **cursiva**.

Cuando el usuario hace clic en cualquiera de los objetos **JCheckBox**, **negrita** o **cursiva**, se llama al método **itemStateChanged** (línea 55). Este método utiliza a **e.getSource()** para determinar en cuál objeto **JCheckBox** se hizo clic. Si fue en el objeto **negrita** de **JCheckBox**, la estructura **if/else** de las líneas 58 a 61

```
if ( e.getStateChange() == ItemEvent.SELECTED )
   valNegrita = Font.BOLD;
else
   valNegrita = Font.PLAIN;
```

utiliza el método ***getStateChange*** de **ItemEvent** para determinar el estado del botón (**ItemEvent.SELECTED** o **ItemEvent.DESELECTED**). Si se selecciona el estado **negrita**, a la variable entera **valNegrita** se le asigna **Font.BOLD**; de lo contrario, a **valNegrita** se le asigna **Font.PLAIN**. Si hace clic en el objeto **cursiva** de **JCheckBox**, se ejecuta una estructura **if/else** similar. Si se selecciona el estado **cursiva**, a la variable entera **valCursiva** se le asigna **Font.ITALIC**; de lo contrario, a **valCursiva** se le asigna **Font.PLAIN**. La suma de **valNegrita** y **valCursiva** se utiliza en las líneas 69 y 70 como el estilo del nuevo tipo de fuente para el objeto **JTextField**.

## 29.9 JComboBox

Un *cuadro combinado* (algunas veces conocido como *lista desplegable*) proporciona una lista de elementos, de los cuales el usurario puede seleccionar uno. Los cuadros combinados se implementan mediante la clase ***JComboBox***, la cual hereda de la clase **JComponent**. Los objetos **JComboBox** generan eventos **ItemEvent**, al igual que los objetos **JCheckBox** y **JRadioButton**.

La aplicación de la figura 29.13 utiliza un objeto **JComboBox** para proporcionar una lista de cuatro nombres de archivos de imágenes. Al seleccionar un archivo de imagen, la imagen correspondiente aparece como un icono en un objeto **JLabel**. En las capturas de las imágenes de este programa aparece la lista **JComboBox** después de haber hecho la selección, para ilustrar cuál nombre de archivo de imagen se seleccionó.

```
// Figura 29.13: PruebaCuadroCombinado.java
// Uso de un objeto JComboBox para seleccionar una imagen a desplegar.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PruebaCuadroCombinado extends JFrame {
   private JComboBox imagenes;
   private JLabel etiqueta;
   private String nombres[] =
      { "insecto1.gif", "insecto2.gif",
        "insectoviaje.gif", "insectanim.gif" };
   private Icon iconos[] =
      { new ImageIcon( nombres[ 0 ] ),
        new ImageIcon( nombres[ 1 ] ),
        new ImageIcon( nombres[ 2 ] ),
        new ImageIcon( nombres[ 3 ] ) };

   public PruebaCuadroCombinado()
   {
      super( "Prueba de JComboBox" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      imagenes = new JComboBox( nombres );
      imagenes.setMaximumRowCount( 3 );

      imagenes.addItemListener(
         new ItemListener() {
            public void itemStateChanged( ItemEvent e )
            {
              etiqueta.setIcon(
```

**Figura 29.13** Programa que utiliza un objeto **JComboBox** para seleccionar un icono; **PruebaCuadroCombinado.java**. (Parte 1 de 2.)

```
34                      iconos[ imagenes.getSelectedIndex() ] );
35                } // fin del método itemStateChanged
36             } // fin de la clase interna anónima
37          ); // fin de addItemListener
38
39          c.add( imagenes );
40
41          etiqueta = new JLabel( iconos[ 0 ] );
42          c.add( etiqueta );
43
44          setSize( 350, 100 );
45          show();
46       } // fin del constructor de PruebaCuadroCombinado
47
48       public static void main( String args[] )
49       {
50          PruebaCuadroCombinado ap = new PruebaCuadroCombinado();
51
52          ap.addWindowListener(
53             new WindowAdapter() {
54                public void windowClosing( WindowEvent e )
55                {
56                   System.exit( 0 );
57                } // fin del método windowClosing
58             } // fin de la clase interna anónima
59          ); // fin de addWindowListener
60       } // fin de main
61    } // fin de la clase PruebaCuadroCombinado
```

Prueba de JComboBox
insecto1.gif
insecto1.gif
insecto2.gif
insectoviaje.gif

Prueba de JComboBox
insecto2.gif
insecto1.gif
insecto2.gif
insectoviaje.gif

Una **barra de desplazamiento** para desplazarse a través de los elementos de la lista

flechas de desplazamiento

cuadro de desplazamiento

Prueba de JComboBox
insectoviaje.gif
insecto2.gif
insectoviaje.gif
insectanim.gif

Prueba de JComboBox
insectanim.gif
insecto2.gif
insectoviaje.gif
insectanim.gif

**Figura 29.13** Programa que utiliza un objeto **JComboBox** para seleccionar un icono; **PruebaCuadroCombinado.java**. (Parte 2 de 2.)

Las líneas 13 a 17

```
private Icon iconos[] =
   {  new ImageIcon( nombres[ 0 ] ),
      new ImageIcon( nombres[ 1 ] ),
      new ImageIcon( nombres[ 2 ] ),
      new ImageIcon( nombres[ 3 ] ) };
```

declaran e inicializan el arreglo **`iconos`** con cuatro nuevos objetos **`ImageIcon`**. El arreglo **`String`** llamado **`nombres`** (definido en las líneas 10 a 12) contiene los nombres de los cuatro archivos de imágenes que están almacenados en el mismo directorio que la aplicación.

La línea 26

```
imagenes = new JComboBox( nombres );
```

crea un **`JComboBox`**, utilizando los **`Strings`** del arreglo **`nombres`** como elementos para la lista. Un *índice* numérico lleva el registro del orden de los elementos del objeto **`JComboBox`**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento agregado a un objeto **`JComboBox`** aparece como el elemento actualmente seleccionado, cuando el objeto **`JComboBox`** aparece en pantalla. Otros elementos se seleccionan haciendo clic en el objeto **`JComboBox`**. Cuando se hace clic en este objeto, el **`JComboBox`** se expande en una lista, en la que el usuario puede seleccionar un elemento.

La línea 27

```
imagenes.setMaximumRowCount( 3 );
```

utiliza el método ***`setMaximumRowCount`*** de **`JComboBox`** para establecer el número máximo de elementos que se muestran cuando el usuario hace clic en el objeto **`JComboBox`**. Si hay más elementos en el objeto **`JComboBox`** que el número máximo de elementos que aparecen en pantalla, el objeto **`JComboBox`** proporciona automáticamente una *barra de desplazamiento* (vea la primera imagen capturada en pantalla), la cual permite al usuario ver todos los elementos de la lista. El usuario puede hacer clic en las *flechas de desplazamiento* en la parte superior e inferior de la barra de desplazamiento, para moverse hacia arriba y hacia abajo por la lista, un elemento a la vez, o puede arrastrar el *cuadro de desplazamiento* (que está en medio de la barra de desplazamiento) hacia arriba o hacia abajo para avanzar por la lista. Para arrastrar este cuadro de desplazamiento, mantenga oprimido el botón del ratón con su puntero en el cuadro de desplazamiento, y después mueva el ratón.

**Observación de apariencia visual 29.9**

*Establezca el conteo máximo de filas para un objeto **`JComboBox`** en un número que evite que la lista se expanda más allá de los límites de la ventana o del applet en que se utilice. Esto garantizará que la lista aparezca correctamente cuando el usuario la expanda.*

Las líneas 29 a 37

```
imagenes.addItemListener(
   new ItemListener() {
      public void itemStateChanged( ItemEvent e )
      {
        etiqueta.setIcon(
              iconos[ imagenes.getSelectedIndex() ] );
      } // fin del método itemStateChanged
   } // fin de la clase interna anónima
); // fin de addItemListener
```

registran una instancia de una clase interna anónima que implementa a **`ItemListener`** como el componente de escucha para el objeto **`imagenes`** de **`JComboBox`**. Cuando el usuario hace una selección de **`imagenes`**, el método **`itemStateChanged`** (línea 31) establece el objeto **`Icon`** para **`etiqueta`**. Para seleccionar el objeto **`Icon`** del arreglo **`iconos`**, se determina el número del índice del elemento seleccionado en el objeto **`JComboBox`** con el método ***`getSelectedIndex`*** de la línea 34.

## 29.10 Manejo de eventos de ratón

En esta sección presentaremos las interfaces que escuchan eventos ***`MouseListener`*** y ***`MouseMotionListener`*** para manejar *eventos del ratón*. Estos eventos pueden ser atrapados por cualquier componente GUI que se derive de **`java.awt.Component`**. La figura 29.14 resume los métodos de las interfaces **`MouseListener`** y **`MouseMotionListener`**.

| **Métodos de las interfaces `MouseListener` y `MouseMotionListener`** |
|---|
| **`public void mousePressed( MouseEvent e )                // MouseListener`** |
| Se llama cuando se oprime un botón del ratón y el puntero de éste se encuentra sobre un componente. |
| **`public void mouseClicked( MouseEvent e )                // MouseListener`** |
| Se llama cuando se oprime y se suelta un botón del ratón en un componente, sin mover el cursor del ratón. |
| **`public void mouseReleased( MouseEvent e )               // MouseListener`** |
| Se llama cuando se suelta un botón del ratón, después de oprimirlo. Este evento siempre va después de un evento **`mousePressed`**. |
| **`public void mouseEntered( MouseEvent e )                // MouseListener`** |
| Se llama cuando el cursor del ratón entra a los límites de un componente. |
| **`public void mouseExited( MouseEvent e )                 // MouseListener`** |
| Se llama cuando el cursor del ratón sale de los límites de un componente. |
| **`public void mouseDragged( MouseEvent e )                // MouseMotionListener`** |
| Se llama cuando se oprime el botón del ratón y éste se mueve. Este evento siempre va después de una llamada a **`mousePressed`**. |
| **`public void mouseMoved( MouseEvent e )                  // MouseMotionListener`** |
| Se llama cuando el ratón se mueve y el cursor del éste se encuentra sobre un componente. |

**Figura 29.14** Métodos de las interfaces **`MouseListener`** y **`MouseMotionListener`**.

Cada uno de los métodos manejadores de eventos de ratón toma un objeto ***`MouseEvent`*** como su argumento. Un objeto **`MouseEvent`** contiene información acerca del evento de ratón que ocurrió, incluso las coordenadas *x* y *y* de la posición en la que ocurrió el evento. Los métodos de **`MouseListener`** y **`MouseMotionListener`** se llaman siempre que el ratón interactúa con un objeto **`Component`**, si hay objetos componentes de escucha registrados para un objeto **`Component`** específico. El método ***`mousePressed`*** se llama cuando se oprime un botón del ratón y el cursor de éste se encuentra sobre un componente. Por medio de los métodos y constantes de la clase ***`InputEvent`*** (la superclase de **`MouseEvent`**), un programa puede determinar cuál fue el botón que oprimió el usuario. El método ***`mouseClicked`*** se llama siempre que se suelta un botón del ratón sin moverlo, después de una operación **`mousePressed`**. El método ***`mouseReleased`*** se llama siempre que se suelta un botón del ratón. El método ***`mouseEntered`*** se llama cuando el cursor del ratón entra a los límites físicos de un objeto **`Component`**. El método ***`mouseExited`*** se llama cuando el cursor del ratón sale de los límites físicos de un objeto **`Component`**. El método ***`mouseDragged`*** se llama cuando el botón del ratón se oprime y se suelta, y el ratón se mueve (un proceso conocido como *arrastrar*). El evento **`mouseDragged`** va después de un evento ***`mousePressed`*** y antes de un evento ***`mouseReleased`***. El método ***`mouseMoved`*** se llama cuando el ratón se mueve y el cursor del ratón está sobre un componente (y los botones del ratón no están oprimidos).

### Observación de apariencia visual 29.10

*Las llamadas al método **`mouseDragged`** se envían al objeto **`MouseMotionListener`** para el objeto **`Component`** en el que se inició la operación de arrastre. De manera similar, la llamada al método **`mouseReleased`** se envía al objeto **`MouseListener`** para el objeto **`Component`** en el que se inició la operación de arrastre.*

La aplicación **`RastreadorRaton`** (figura 29.15) muestra el uso de los métodos **`MouseListener`** y **`MouseMotionListener`**. La clase de la aplicación implementa ambas interfaces, de manera que pueda escuchar sus propios eventos de ratón. Observe que el programador debe definir los siete métodos de estas dos interfaces cuando una clase implementa ambas interfaces.

```
// Figura 29.15: RastreadorRaton.java
// Demostración de los eventos de ratón.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RastreadorRaton extends JFrame
              implements MouseListener, MouseMotionListener {
   private JLabel barraEstado;

   public RastreadorRaton()
   {
      super( “Demostracion de los eventos de raton” );

      barraEstado = new JLabel();
      getContentPane().add( barraEstado, BorderLayout.SOUTH );

      // la aplicación escucha sus propios eventos de ratón
      addMouseListener( this );
      addMouseMotionListener( this );

      setSize( 275, 100 );
      show();
   } // fin del constructor de RastreadorRaton

   // Manejadores de eventos de MouseListener
   public void mouseClicked( MouseEvent e )
   {
      barraEstado.setText( “Se hizo clic en [“ + e.getX() +
                           “, “ + e.getY() + “]” );
   } // fin del método mouseClicked

   public void mousePressed( MouseEvent e )
   {
      barraEstado.setText( “Se oprimio en [“ + e.getX() +
                           “, “ + e.getY() + “]” );
   } // fin del método mousePressed

   public void mouseReleased( MouseEvent e )
   {
      barraEstado.setText( “Se solto en [“ + e.getX() +
                           “, “ + e.getY() + “]” );
   } // fin del método mouseReleased

   public void mouseEntered( MouseEvent e )
   {
      barraEstado.setText( “Raton dentro de la ventana” );
   } // fin del método mouseEntered

   public void mouseExited( MouseEvent e )
   {
      barraEstado.setText( “Raton fuera de la ventana” );
```

**Figura 29.15** Demostración del manejo de eventos de ratón; **`RastreadorRaton.java`**. (Parte 1 de 2.)

```
54       } // fin del método mouseExited
55
56       // Manejadores de eventos de MouseMotionListener
57       public void mouseDragged( MouseEvent e )
58       {
59          barraEstado.setText( "Se arrastro en [" + e.getX() +
60                               ", " + e.getY() + "]" );
61       } // fin del método mouseDragged
62
63       public void mouseMoved( MouseEvent e )
64       {
65          barraEstado.setText( "Se movio en [" + e.getX() +
66                               ", " + e.getY() + "]" );
67       } // fin del método mouseMoved
68
69       public static void main( String args[] )
70       {
71          RastreadorRaton ap = new RastreadorRaton();
72
73          ap.addWindowListener(
74             new WindowAdapter() {
75                public void windowClosing( WindowEvent e )
76                {
77                   System.exit( 0 );
78                } // fin del método windowClosing
79             } // fin de la clase interna anónima
80          ); // fin de addWindowListener
81       } // fin de main
82    } // fin de la clase RastreadorRaton
```

Demostracion de los eventos d...
Raton fuera de la ventana

Demostracion de los eventos d...
Raton dentro de la ventana

Demostracion de los eventos d...
Se movio en [89, 31]

Demostracion de los eventos d...
Se oprimio en [127, 47]

Demostracion de los eventos d...
Se arrastro en [171, 43]

Demostracion de los eventos d...
Se solto en [158, 47]

Demostracion de los eventos d...
Se hizo clic en [4, 26]

**Figura 29.15** Demostración del manejo de eventos de ratón; **`RastreadorRaton.java`**. (Parte 2 de 2.)

Cada evento de ratón hace que aparezca un objeto **`String`** en el objeto **`barraEstado`** de **`JLabel`**, que se encuentra en la parte inferior de la ventana.

Las líneas 16 y 17 del constructor

```
barraEstado = new JLabel();
getContentPane().add( barraEstado, BorderLayout.SOUTH );
```

definen el objeto **barraEstado** de **JLabel** y lo adjuntan al panel de contenido. Hasta el momento, cada vez que utilizábamos el panel de contenido, se hacía una llamada al método **setLayout** para establecer en **FlowLayout** al administrador de diseño del panel de contenido. Esto permitía al panel de contenido mostrar de izquierda a derecha los componentes GUI que le íbamos adjuntando. Si los componentes GUI no caben en una sola línea, el diseño **FlowLayout** crea líneas adicionales para seguir mostrando los componentes GUI. En realidad, el administrador de diseño predeterminado es ***BorderLayout***, el cual divide el área del panel de contenido en cinco regiones: norte, sur, este, oeste y centro. La línea 17 utiliza una nueva versión del método **add** de **Container** para adjuntar **barraEstado** a la región sur (***BorderLayout.SOUTH***), la cual se extiende a lo largo de toda la parte inferior del panel de contenido. Más adelante en este capítulo describiremos detalladamente el uso de **BorderLayout** y varios otros administradores de diseño.

Las líneas 20 y 21 del constructor

```
addMouseListener( this );
addMouseMotionListener( this );
```

registran el objeto de ventana **RastreadorRaton** como el componente que escucha sus propios eventos de ratón. Los métodos ***addMouseListener*** y ***addMouseMotionListener*** son métodos de **Component** que pueden utilizarse para registrar componentes para escuchar eventos de ratón de un objeto de cualquier clase que extienda a **Component**.

Cuando el ratón entra o sale del área de la aplicación, se llama a los métodos **mouseEntered** (línea 46) y **mouseExited** (línea 51), respectivamente. Ambos métodos muestran un mensaje en la **barraEstado**, el cual indica que el ratón se encuentra dentro de la aplicación, o que está fuera de ella (vea las primeras dos capturas de imágenes de pantalla).

Cuando ocurre cualquiera de los otros cinco eventos, aparece un mensaje en la **barraEstado** que incluye un objeto **String**, el cual representa el evento que ocurrió y las coordenadas en donde ocurrió ese evento de ratón. Las coordenadas *x* y *y* del ratón, al momento en que ocurrió el evento, se obtienen mediante los métodos ***getX*** y ***getY*** de **MouseEvent**, respectivamente.

## 29.11 Administradores de diseño

Los *administradores de diseño* ordenan los componentes GUI en un contenedor, para fines de presentación. Los administradores de diseño proporcionan herramientas de diseño básicas, que son más fáciles de utilizar que determinar la posición y el tamaño exactos de cada componente GUI. Esto permite al programador concentrarse en la "apariencia visual" básica, para dejar a los administradores de diseño que procesen la mayor parte de los detalles de diseño.

**Observación de apariencia visual 29.11**

*La mayoría de los entornos de programación de Java proporcionan herramientas de diseño GUI, las cuales ayudan a un programador a diseñar de manera gráfica una GUI, y después escriben automáticamente el código de Java necesario para crear la GUI.*

Algunos diseñadores de GUIs también permiten al programador utilizar los administradores de diseño que describimos aquí. La figura 29.16 sintetiza los administradores de diseño que presentamos en este capítulo.

| Administrador de diseño | Descripción |
|---|---|
| **FlowLayout** | Es el predeterminado para **java.awt.Applet**, **java.awt.Panel** y **javax.swing.Panel**. Coloca los componentes secuencialmente (de izquierda a derecha) en el orden en el que se agregaron. También es posible especificar el orden de los componentes utilizando el método **add** de **Container**, el cual toma como argumentos un objeto **Component** y un valor entero que representa la posición del índice. |
| **BorderLayout** | Es el predeterminado para los paneles de contenido de objetos **JFrame** (y otras ventanas) y **Japplet**. Ordena los componentes en cinco áreas: Norte, Sur, Este, Oeste y Centro. |
| **GridLayout** | Ordena los componentes en filas y columnas. |

**Figura 29.16** Administradores de diseño.

La mayoría de los ejemplos anteriores de applets y aplicaciones en los que creamos nuestra propia GUI utilizan el administrador de diseño ***FlowLayout***. La clase ***FlowLayout*** hereda de la clase **Object** e implementa la interfaz ***LayoutManager***, la cual define los métodos que utiliza un administrador de diseño para ordenar y ajustar el tamaño de los componentes GUI en un contenedor.

### 29.11.1 FlowLayout

Éste es el administrador de diseño más básico. Los componentes GUI se colocan en un contenedor de izquierda a derecha, en el orden en el que se agregan al contenedor. Al llegar al límite del contenedor, los componentes continúan en la siguiente línea. La clase **FlowLayout** permite que los componentes GUI estén *alineados a la izquierda*, *centrados* (la opción predeterminada) y *alineados a la derecha*.

La aplicación de la figura 29.17 crea tres objetos **JButton** y los agrega a la aplicación utilizando el administrador de diseño **FlowLayout**. Los componentes se alinean automáticamente al centro. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación al centro. Cada botón tiene su propio manejador de eventos que se define mediante una clase interna que implementa a **ActionListener**.

```
// Figura 29.17: DemoFlowLayout.java
// Demostración de las alineaciones de FlowLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoFlowLayout extends JFrame {
   private JButton izquierda, centro, derecha;
   private Container c;
   private FlowLayout disenio;

   public DemoFlowLayout()
   {
      super( “Demo de FlowLayout” );

      disenio = new FlowLayout();

      c = getContentPane();
      c.setLayout( disenio );

      izquierda = new JButton( “Izquierda” );
      izquierda.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               disenio.setAlignment( FlowLayout.LEFT );

               // vuelve a alinear los componentes adjuntos
               disenio.layoutContainer( c );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( izquierda );
```

**Figura 29.17** Programa que demuestra el uso de componentes en **FlowLayout**; **DemoFlowLayout.java**. (Parte 1 de 2.)

```

      centro = new JButton( “Centro” );
      centro.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               disenio.setAlignment( FlowLayout.CENTER );

               // volver a alinear los componentes adjuntos
               disenio.layoutContainer( c );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( centro );

      derecha = new JButton( “Derecha” );
      derecha.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               disenio.setAlignment( FlowLayout.RIGHT );

               // vuelve a alinear los componentes adjuntos
               disenio.layoutContainer( c );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      c.add( derecha );

      setSize( 300, 75 );
      show();
   } // fin del constructor de DemoFlowLayout

   public static void main( String args[] )
   {
      DemoFlowLayout ap = new DemoFlowLayout();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase DemoFlowLayout
```

**Figura 29.17** Programa que demuestra el uso de componentes en **`FlowLayout`**; **`DemoFlowLayout.java`**. (Parte 2 de 2.)

Como vimos anteriormente, el diseño de un contenedor se establece mediante el método ***setLayout*** de la clase **Container**. La línea 19

```
c.setLayout( disenio );
```

establece el administrador de diseño del panel de contenido al **FlowLayout** definido en la línea 16. En general, el diseño se establece antes de agregar cualquier componente GUI a un contenedor.




**Observación de apariencia visual 29.12**

*Cada contenedor puede tener solamente un administrador de diseño a la vez (varios contenedores en el mismo programa pueden tener distintos administradores de diseño).*

El manejador de eventos **actionPerformed** de cada botón ejecuta dos instrucciones. Por ejemplo, la línea 26 del método **actionPerformed** del botón **izquierda**

```
disenio.setAlignment( FlowLayout.LEFT );
```

utiliza el método ***setAlignment*** de **FlowLayout** para cambiar la alineación de **FlowLayout** a la izquierda (***FlowLayout.LEFT***). La línea 29

```
disenio.layoutContainer( c );
```

utiliza el método ***LayoutContainer*** de la interfaz **LayoutManager** para especificar que el panel de contenido debe volver a ordenarse con base en el diseño ajustado.

De acuerdo con el botón en el que se haya hecho clic, el método **actionPerformed** de cada botón establece la alineación de **FlowLayout** a ***FlowLayout.LEFT***, ***FlowLayout.CENTER*** o ***FlowLayout.RIGHT***.

## 29.11.2 BorderLayout

El administrador de diseño ***BorderLayout*** (el predeterminado para el panel de contenido) ordena los componentes en cinco regiones: *Norte*, *Sur*, *Este*, *Oeste* y *Centro* (la región Norte corresponde a la parte superior del contenedor). La clase **BorderLayout** hereda de **Object** e implementa la interfaz ***LayoutManager2*** (una subinterfaz de **LayoutManager** que agrega varios métodos para mejorar el procesamiento de los diseños).

Es posible agregar hasta cinco componentes directamente a un diseño **BorderLayout**; uno para cada región. Los componentes que se colocan en las regiones Norte y Sur se extienden horizontalmente hacia los lados del contenedor, y su altura depende de los componentes que se coloquen en esas regiones. Las regiones Este y Oeste se expanden verticalmente entre las regiones Norte y Sur, y su ancho depende de los componentes que se coloquen en esas regiones. El componente colocado en la región Centro se expande para ocupar todo el espacio restante en el diseño (ésta es la razón por la que el objeto **JTextArea** de la figura 29.18 ocupa la ventana completa). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con componentes GUI. Si la región Norte o la región Sur no están ocupadas, los componentes GUI de las regiones Este, Centro y Oeste se expanden verticalmente para llenar el espacio restante. Si la región Este o la región Oeste no están ocupadas, el componente GUI de la región Centro se expande horizontalmente para llenar el espacio restante. Si la región Centro no está ocupada, el área se deja vacía; los demás componentes GUI no se expanden para llenar el espacio restante.

La aplicación de la figura 29.18 demuestra el uso del administrador de diseño **BorderLayout** con cinco objetos **JButton**.

```
// Figura 29.18: DemoBorderLayout.java
// Demostración de BorderLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

**Figura 29.18** Demostración del uso de componentes en **BorderLayout**; **DemoBorderLayout.java**. (Parte 1 de 3.)

```
public class DemoBorderLayout extends JFrame
                              implements ActionListener {
   private JButton b[];
   private String nombres[] =
      { “Ocultar Norte”, “Ocultar Sur”, “Ocultar Este”,
        “Ocultar Oeste”, “Ocultar Centro” };
   private BorderLayout disenio;

   public DemoBorderLayout()
   {
      super( “Demo de BorderLayout” );

      disenio = new BorderLayout( 5, 5 );

      Container c = getContentPane();
      c.setLayout( disenio );

      // instanciar objetos botón
      b = new JButton[ nombres.length ];

      for ( int i = 0; i < nombres.length; i++ ) {
         b[ i ] = new JButton( nombres[ i ] );
         b[ i ].addActionListener( this );
      } // fin de for

      // el orden no importa
      c.add( b[ 0 ], BorderLayout.NORTH );  // Posición Norte
      c.add( b[ 1 ], BorderLayout.SOUTH );  // Posición Sur
      c.add( b[ 2 ], BorderLayout.EAST );   // Posición Este
      c.add( b[ 3 ], BorderLayout.WEST );   // Posición Oeste
      c.add( b[ 4 ], BorderLayout.CENTER ); // Posición Centro

      setSize( 400, 250 );
      show();
   } // fin del constructor DemoBorderLayout

   public void actionPerformed( ActionEvent e )
   {
      for ( int i = 0; i < b.length; i++ )
         if ( e.getSource() == b[ i ] )
            b[ i ].setVisible( false );
         else
            b[ i ].setVisible( true );

      // reajusta el diseño del panel de contenido
      disenio.layoutContainer( getContentPane() );
   } // fin del método actionPerformed

   public static void main( String args[] )
   {
      DemoBorderLayout ap = new DemoBorderLayout();

      ap.addWindowListener(
         new WindowAdapter() {
```

**Figura 29.18** Demostración del uso de componentes en **`BorderLayout`**; **`DemoBorderLayout.java`**. (Parte 2 de 3.)

```
61                public void windowClosing( WindowEvent e )
62                {
63                   System.exit( 0 );
64                } // fin del método windowClosing
65             } // fin de la clase interna anónima
66          ); // fin de addWindowListener
67       } // fin de main
68    } // fin de la clase DemoBorderLayout
```

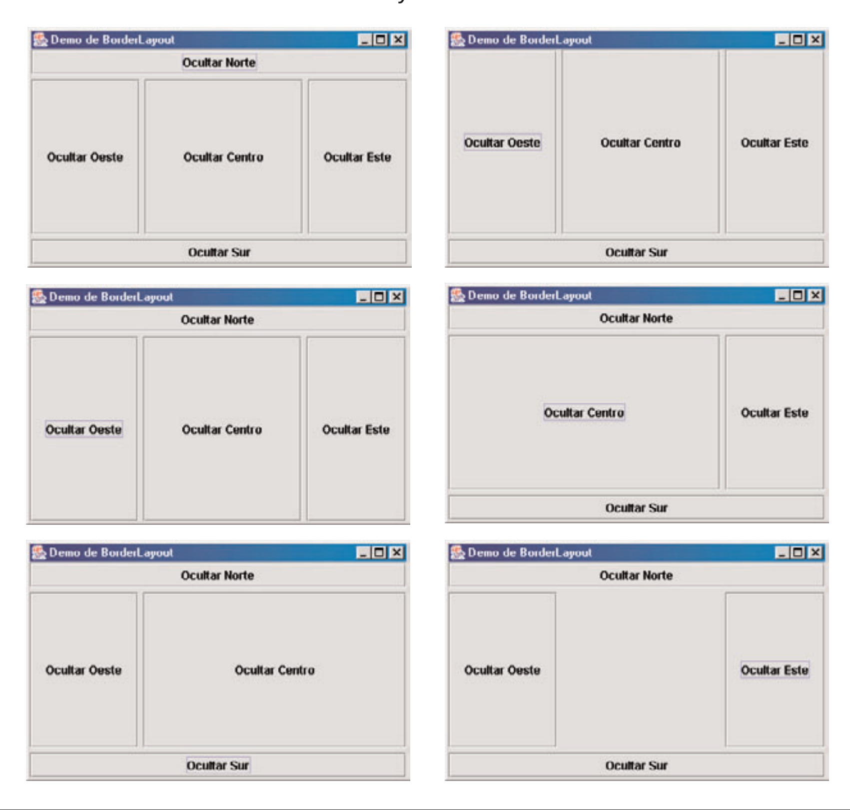



**Figura 29.18** Demostración del uso de componentes en **`BorderLayout`**; **`DemoBorderLayout.java`**. (Parte 3 de 3.)

La línea 19 del constructor

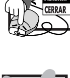
```
disenio = new BorderLayout( 5, 5 );
```

define un diseño **`BorderLayout`**. Los argumentos especifican el número de pixeles entre componentes ordenados horizontalmente (*espacio libre horizontal*) y el número de pixeles entre componentes ordenados verticalmente (*espacio libre vertical*), respectivamente. El constructor predeterminado de **`BorderLayout`** proporciona 0 pixeles de espacio libre horizontal y vertical. La línea 22 utiliza el método **`setLayout`** para establecer el diseño del panel de contenido a **`disenio`**.

Para agregar objetos **`Component`** a un diseño **`BorderLayout`** se requiere un método **`add`** distinto de la clase **`Container`**, el cual toma dos argumentos: el objeto **`Component`** que va a agregarse y la región en la que se colocará este objeto. Por ejemplo, la línea 33

```
add( b[ 0 ], BorderLayout.NORTH );  // Posición Norte
```

especifica que el componente **`b[0]`** va a colocarse en la posición **`NORTH`**. Los componentes pueden agregarse en cualquier orden, pero solamente puede agregarse un componente a cada región.



**Observación de apariencia visual 29.13**

*Si no se especifica una región al agregar un objeto* **`Component`** *a un diseño* **`BorderLayout`**, *se asume que el objeto* **`Component`** *va a agregarse a la región* **`BorderLayout.CENTER`**.

**Error común de programación 29.5**

*Si se agrega más de un componente a una región específica en un diseño* **`BorderLayout`**, *sólo se desplegará el último componente que se haya agregado. No hay un mensaje de error para indicar este problema.*

Cuando el usuario hace clic en un objeto **`JButton`** específico del diseño, se hace una llamada al método **`actionPerformed`** (línea 43). El ciclo **`for`** de la línea 46 utiliza la siguiente estructura **`if/else`**

```
if ( e.getSource() == b[ i ] )
   b[ i ].setVisible( false );
else
   b[ i ].setVisible( true );
```

para ocultar el objeto **`JButton`** específico que generó el evento. El método ***`setVisible`*** (heredado por **`JButton`** de la clase **`Component`**) se llama con un argumento **`false`** para ocultar el objeto **`JButton`**. Si el objeto **`JButton`** actual del arreglo no es el que generó el evento, se hace una llamada al método **`setVisible`** con un argumento **`true`** para garantizar que el objeto **`JButton`** se despliegue en la pantalla. La línea 52

```
disenio.layoutContainer( getContentPane() );
```

utiliza el método **`layoutContainer`** de **`LayoutManager`** para recalcular el diseño del panel de contenido. Observe en las capturas de pantalla de la figura 29.18 que ciertas regiones del diseño **`BorderLayout`** cambian de forma, a medida que se ocultan y se despliegan los objetos **`JButton`** en otras regiones. Intente cambiar el tamaño de la ventana de la aplicación para que vea cómo se ajusta el tamaño de las diversas regiones, con base en el ancho y la altura de la ventana.

## 29.11.3 GridLayout

El administrador de diseño ***`GridLayout`*** divide el contenedor en una cuadrícula, de manera que los componentes puedan colocarse en filas y columnas. La clase **`GridLayout`** hereda directamente de la clase **`Object`** e implementa la interfaz **`LayoutManager`**. Cada objeto **`Component`** de un diseño **`GridLayout`** tiene el mismo ancho y alto. Los componentes se agregan a un diseño **`GridLayout`** a partir de la celda superior izquierda de la cuadrícula, y continúan agregándose de izquierda a derecha hasta que la fila se llena. Después, el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. La figura 29.19 muestra el uso del administrador de diseño **`GridLayout`** con seis objetos **`JButton`**.

```
// Figura 29.19: DemoGridLayout.java
// Demostración de GridLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoGridLayout extends JFrame
                            implements ActionListener {
   private JButton b[];
   private String nombres[] =
      { “uno”, “dos”, “tres”, “cuatro”, “cinco”, “seis” };
   private boolean alternar = true;
   private Container c;
   private GridLayout cuadricula1, cuadricula2;

   public DemoGridLayout()
```

**Figura 29.19** Programa que muestra el uso de componentes en **`GridLayout`**; **`DemoGridLayout.java`**. (Parte 1 de 2.)

```
   {
      super( "Demostracion de GridLayout" );

      cuadricula1 = new GridLayout( 2, 3, 5, 5 );
      cuadricula2 = new GridLayout( 3, 2 );

      c = getContentPane();
      c.setLayout( cuadricula1 );

      // crea y agrega botones
      b = new JButton[ nombres.length ];

      for (int i = 0; i < nombres.length; i++ ) {
         b[ i ] = new JButton( nombres[ i ] );
         b[ i ].addActionListener( this );
         c.add( b[ i ] );
      }

      setSize( 300, 150 );
      show();
   } // fin del constructor de DemoGridLayout

   public void actionPerformed( ActionEvent e )
   {
      if ( alternar )
         c.setLayout( cuadricula2 );
      else
         c.setLayout( cuadricula1 );

      alternar = !alternar;
      c.validate();
   } // fin del método actionPerformed

   public static void main( String args[] )
   {
      DemoGridLayout ap = new DemoGridLayout();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
     ); // fin de addWindowListener
   } // fin de main
} // fin de la clase DemoGridLayout
```

**Figura 29.19** Programa que muestra el uso de componentes en **`GridLayout`**; **`DemoGridLayout.java`**. (Parte 2 de 2.)

Las líneas 20 y 21 del constructor

```
cuadricula1 = new GridLayout( 2, 3, 5, 5 );
cuadricula2 = new GridLayout( 3, 2 );
```

definen dos objetos **`GridLayout`**. El constructor de **`GridLayout`** utilizado en la línea 20 especifica un objeto **`GridLayout`** con **`2`** filas, **`3`** columnas, **`5`** pixeles de espacio libre horizontal entre los objetos **`Component`** de la cuadrícula, y **`5`** pixeles de espacio libre vertical entre los objetos **`Component`** de la cuadrícula. El constructor de **`GridLayout`** utilizado en la línea 21 especifica un objeto **`GridLayout`** con **`3`** filas, **`2`** columnas y nada de espacio libre.

Los objetos **`JButton`** de este ejemplo se ordenan inicialmente por medio de **`cuadricula1`** (que se establece para el panel de contenido en la línea 24 a través del método **`setLayout`**). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, etcétera. Cuando se oprime un objeto **`JButton`**, se hace una llamada al método **`actionPerformed`** (línea 39). Cada llamada a **`actionPerformed`** cambia el diseño entre **`cuadricula2`** y **`cuadricula1`**.

La línea 47

```
c.validate();
```

muestra una manera de redistribuir un contenedor que haya cambiado su diseño. El método ***`validate`*** de **`Container`** recalcula la distribución del contenedor con base en el administrador de diseño actual para el objeto **`Container`** y el conjunto actual de componentes GUI desplegados en pantalla.

## 29.12 Paneles

Las GUIs complejas (como la figura 29.1) requieren que cada componente se coloque en una ubicación exacta. A menudo, éstas consisten en varios *paneles* en los que cada componente está ordenado con un diseño específico. Los paneles se crean mediante la clase ***`Jpanel`*** (una subclase de **`JComponent`**). La clase **`JComponent`** hereda de **`java.awt.Container`**, por lo que todo **`JPanel`** es un **`Container`**. Por lo tanto, es posible agregar muchos componentes a los objetos **`JPanel`**, incluso otros paneles.

El programa de la figura 29.20 muestra cómo puede utilizarse un objeto **`JPanel`** para crear un diseño más complejo para objetos **`Component`**.

```
// Figura 29.20: DemoPanel.java
// Uso de un objeto JPanel para ayudar a distribuir los componentes en un diseño.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoPanel extends JFrame {
   private JPanel panelBotones;
   private JButton botones[];

   public DemoPanel()
   {
      super( "Demostracion de JPanel" );

      Container c = getContentPane();
      panelBotones = new JPanel();
      botones = new JButton[ 5 ];

      panelBotones.setLayout(
```

**Figura 29.20** Un objeto **`JPanel`** con cinco objetos **`JButton`** en un diseño **`GridLayout`** adjunto a la región **`SOUTH`** de un diseño **`BorderLayout`**; **`DemoPanel.java`**. (Parte 1 de 2.)

```
20            new GridLayout( 1, botones.length ) );
21
22         for ( int i = 0; i < botones.length; i++ ) {
23            botones[ i ] = new JButton( "Boton " + (i + 1) );
24            panelBotones.add( botones[ i ] );
25         }
26
27         c.add( panelBotones, BorderLayout.SOUTH );
28
29         setSize( 425, 150 );
30         show();
31      } // fin del constructor DemoPanel
32
33      public static void main( String args[] )
34      {
35         DemoPanel ap = new DemoPanel();
36
37         ap.addWindowListener(
38            new WindowAdapter() {
39               public void windowClosing( WindowEvent e )
40               {
41                  System.exit( 0 );
42               } // fin del método windowClosing
43            } // fin de la clase interna anónima
44         ); // fin de addWindowListener
45      } // fin de main
46   } // fin de la clase DemoPanel
```

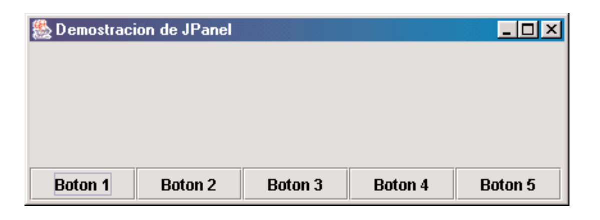



**Figura 29.20** Un objeto **`JPanel`** con cinco objetos **`JButton`** en un diseño **`GridLayout`** adjunto a la región **`SOUTH`** de un diseño **`BorderLayout`**; **`DemoPanel.java`**. (Parte 2 de 2.)

Una vez creado el objeto **`panelBotones`** de **`JPanel`** de la línea 16, las líneas 19 y 20

```
panelBotones.setLayout(
   new GridLayout( 1, botones.length ) );
```

establecen el diseño de **`panelBotones`** en un **`GridLayout`** de una fila y cinco columnas (hay cinco objetos **`JButton`** en el arreglo **`botones`**). Los cinco objetos **`JButton`** del arreglo **`botones`** se agregan al objeto **`JPanel`** en el ciclo de la línea 24, por medio de la instrucción:

```
panelBotones.add( botones[ i ] );
```

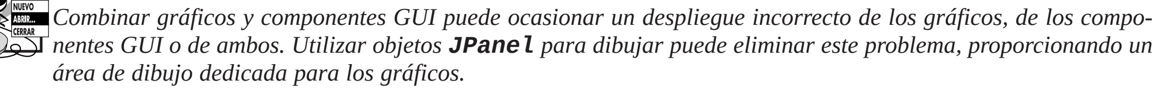
Observe que los botones se agregan directamente al objeto **`JPanel;`** la clase **`JPanel`** no tiene un panel de contenido como el de un applet o un objeto **`JFrame`**. La línea 27

```
c.add( panelBotones, BorderLayout.SOUTH );
```

utiliza el diseño **`BorderLayout`** predeterminado del panel de contenido para agregar **`panelBotones`** a la región **`SOUTH`**. Observe que la altura de esta región se rige por los botones de **`panelBotones`**. Un objeto **`JPanel`** ajusta su tamaño según los componentes que contiene. A medida que se agregan más componentes, el objeto **`JPanel`** crece (de acuerdo con las restricciones de su administrador de diseño) para dar cabida a esos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de diseño afecta al tamaño de los objetos **`JButton`**.

## 29.13 Creación de una subclase autocontenida de `JPanel`

Un objeto **`JPanel`** puede utilizarse como *área de dibujo dedicada*, la cual puede recibir eventos de ratón y, a menudo, se extiende para crear nuevos componentes. En ejercicios anteriores tal vez haya observado que el combinar componentes GUI de Swing con el dibujo en una ventana o subprograma, con frecuencia ocasiona que los componentes GUI o los gráficos se desplieguen en forma incorrecta. Esto se debe a que los componentes GUI de Swing se despliegan utilizando las mismas técnicas de gráficos que los dibujos, y se despliegan en la misma área que los dibujos. El orden en el que se despliegan los componentes GUI y en el que se realiza el dibujo puede ocasionar que se dibuje sobre los componentes GUI, o que los componentes GUI cubran parte de los gráficos. Para solucionar este problema, podemos separar la GUI y los gráficos, creando áreas de dibujo dedicadas como subclases de **`JPanel`**.



#### Observación de apariencia visual 29.14

*Combinar gráficos y componentes GUI puede ocasionar un despliegue incorrecto de los gráficos, de los componentes GUI o de ambos. Utilizar objetos* **`JPanel`** *para dibujar puede eliminar este problema, proporcionando un área de dibujo dedicada para los gráficos.*

Los componentes Swing que heredan de la clase **`JComponent`** contienen el método ***`paintComponent`***, el cual les ayuda a dibujar correctamente dentro del contexto de una GUI de Swing. Al personalizar un objeto **`JPanel`** para usarlo como área de dibujo dedicada, el método **`paintComponent`** debe redefinirse de la siguiente manera:

```
public void paintComponent( Graphics g )
{
   super.paintComponent( g );
   // código adicional para dibujar
}
```

Observe que la llamada a la versión de **`paintComponent`** correspondiente a la superclase aparece como la primera instrucción en el cuerpo del método redefinido. Esto garantiza que la acción de dibujar ocurra en el orden adecuado y que el mecanismo de dibujo de Swing permanezca intacto. Si no se hace la llamada a la versión de **`paintComponent`** correspondiente a la superclase, por lo general lo que ocurre es que el componente GUI personalizado (en este caso, la subclase de **`JPanel`**) no se desplegará apropiadamente en la interfaz de usuario. Además, si se hace la llamada a la versión de la superclase después de ejecutar las instrucciones de dibujo personalizadas, los resultados normalmente se borran.

#### Observación de apariencia visual 29.15

*Cuando se redefine el método* **`paintComponent`** *de un objeto* **`JComponent`***, la primera instrucción del cuerpo siempre debe ser una llamada a la versión original del método de la superclase.*



#### Error común de programación 29.6

*Cuando se redefine el método* **`paintComponent`** *de un objeto* **`JComponent`***, si no se hace una llamada a la versión original de* **`paintComponent`** *de la superclase, el componente GUI no podrá desplegarse apropiadamente en la GUI.*



#### Error común de programación 29.7

*Cuando se redefine el método* **`paintComponent`** *de un objeto* **`JComponent`***, al llamar a la versión original de* **`paintComponent`** *de la superclase después de realizar otro dibujo, se borran los demás dibujos.*

Las clases **`JFrame`** y **`JApplet`** no son subclases de **`JComponent`**; por lo tanto, no contienen el método **`paintComponent`**. Para dibujar directamente en subclases de **`JFrame`** y **`JApplet`**, debe redefinir el método **`paint`**.

#### Observación de apariencia visual 29.16

*Llamar a* **`repaint`** *para un componente GUI de Swing indica que ese componente debe pintarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. La mayoría de los componentes Swing son transparentes de manera predeterminada. Es posible pasar un argumento booleano al método* **`setOpaque`** *de* **`JComponent`** *para indicar si el componente es opaco (***`true`***), o transparente (***`false`***). Los componentes GUI del paquete* **`java.awt`** *son distintos de los componentes Swing en cuanto a que* **`repaint`** *produce una llamada al método* **`update`** *de* **`Component`** *(con lo cual se borra el fondo del componente), y* **`update`***, a su vez, llama al método* **`paint`** *(en lugar de llamar a* **`paintComponent`***).*

Los objetos **JPanel** no generan eventos convencionales como los botones, campos de texto y ventanas, pero son capaces de reconocer eventos de menor nivel, tales como los eventos de ratón y de tecla. El programa de la figura 29.21 permite al usuario dibujar un óvalo en una subclase de **JPanel** con el ratón. La clase **PanelAutoContenido** escucha sus propios eventos de ratón y dibuja un óvalo sobre sí misma. Para determinar la ubicación y el tamaño del óvalo, el usuario debe oprimir el botón del ratón y mantenerlo así, arrastrarlo y soltarlo. La clase **PanelAutoContenido** se encuentra en el paquete **com.deitel.cpec4.cap29**, para poder reutilizarla en el futuro. Por esta razón se importa (mediante la instrucción **import** de la línea 7) a la clase de la aplicación **PanelAutoContenido**.

```
1  // Figura 29.21: PruebaPanelAutoContenido.java
2  // Creación de una subclase autocontenida de JPanel
3  // que procesa sus propios eventos de ratón.
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7  import com.deitel.cpec4.cap29.PanelAutoContenido;
8
9  public class PruebaPanelAutoContenido extends JFrame {
10    private PanelAutoContenido miPanel;
11
12    public PruebaPanelAutoContenido()
13    {
14       miPanel = new PanelAutoContenido();
15       miPanel.setBackground( Color.yellow );
16
17       Container c = getContentPane();
18       c.setLayout( new FlowLayout() );
19       c.add( miPanel );
20
21       addMouseMotionListener(
22          new MouseMotionListener() {
23             public void mouseDragged( MouseEvent e )
24             {
25                setTitle( “Arrastrando: x=” + e.getX() +
26                          “; y=” + e.getY() );
27             } // fin del método mouseDragged
28
29             public void mouseMoved( MouseEvent e )
30             {
31                setTitle( “Moviendo: x=” + e.getX() +
32                          “; y=” + e.getY() );
33             } // fin del método mouseMoved
34          } // fin de la clase interna anónima
35       ); // fin de addMouseMotionListener
36
37       setSize( 300, 200 );
38       show();
39    } // fin del constructor PruebaPanelAutoContenido
40
41    public static void main( String args[] )
42    {
43       PruebaPanelAutoContenido ap =
44          new PruebaPanelAutoContenido();
```

**Figura 29.21** Cómo capturar eventos de ratón con un objeto **JPanel**; **PruebaPanelAutoContenido.java**. (Parte 1 de 2.)

```java

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main
} // fin de la clase PruebaPanelAutoContenido
```

**Figura 29.21** Cómo capturar eventos de ratón con un objeto **JPanel**; **PruebaPanelAutoContenido.java**. (Parte 2 de 2.)

```java
// Figura 29.21: PanelAutoContenido.java
// Una clase JPanel autocontenida que
// maneja sus propios eventos de ratón.
package com.deitel.cpec4.cap29;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PanelAutoContenido extends JPanel {
   private int x1, y1, x2, y2;

   public PanelAutoContenido()
   {
      addMouseListener(
         new MouseAdapter() {
            public void mousePressed( MouseEvent e )
            {
               x1 = e.getX();
               y1 = e.getY();
            } // fin del método mousePressed

            public void mouseReleased( MouseEvent e )
            {
               x2 = e.getX();
               y2 = e.getY();
               repaint();
            } // fin del método mouseReleased
         } // fin de la clase interna anónima
      ); // fin de addMouseListener

      addMouseMotionListener(
         new MouseMotionAdapter() {
            public void mouseDragged( MouseEvent e )
            {
               x2 = e.getX();
               y2 = e.getY();
               repaint();
```

**Figura 29.21** Cómo capturar eventos de ratón con un objeto **JPanel**; **PanelAutoContenido.java**. (Parte 1 de 2.)

```
94             } // fin del método mouseDragged
95          } // fin de la clase interna anónima
96       ); // fin de addMouseMotionListener
97    } // fin del constructor PanelAutoContenido
98
99    public Dimension getPreferredSize()
100   {
101      return new Dimension( 150, 100 );
102   } // fin del método getPreferredSize
103
104   public void paintComponent( Graphics g )
105   {
106      super.paintComponent( g );
107
108      g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
109                  Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
110   } // fin del método paintComponent
111 } // fin de la clase PanelAutoContenido
```

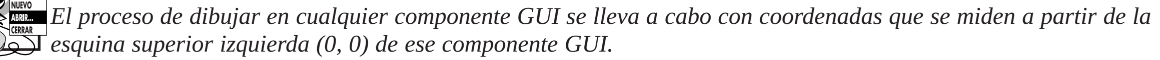
**Figura 29.21** Cómo capturar eventos de ratón con un objeto **JPanel**; **PanelAutoContenido.java**. (Parte 2 de 2.)

El método constructor (línea 12) de la clase de la aplicación **PruebaPanelAutoContenido** crea una instancia de la clase **PanelAutoContenido** y establece en amarillo el color de fondo del **PanelAutoContenido**, de manera que su área sea visible y contraste con el fondo de la ventana de la aplicación.

Para que podamos demostrar la diferencia entre los eventos de movimiento del ratón en el **PanelAutoContenido** y los eventos de movimiento del ratón en la ventana de la aplicación, las líneas 21 a 35 crean una clase interna anónima para manejar los eventos de movimiento del ratón en la aplicación. Los manejadores de eventos **mouseDragged** y **mouseMoved** utilizan el método **setTitle** (heredado de la clase **java.awt.Frame**) para mostrar un objeto **String** en la barra de título de la ventana, e indican las coordenadas *x* y *y* en donde ocurrió el evento de movimiento del ratón.

La clase **PanelAutoContenido** (líneas 65 a 111) extiende a la clase **JPanel**. Las variables de instancia **x1** y **y1** almacenan las coordenadas iniciales en donde ocurre el evento **mousePressed** en el **PanelAutoContenido**. Las variables de instancia **x2** y **y2** almacenan las coordenadas en donde el usuario arrastra el ratón o suelta el botón de éste. Todas las coordenadas son con respecto a la esquina superior izquierda del **PanelAutoContenido**.

### Observación de apariencia visual 29.17



*El proceso de dibujar en cualquier componente GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0, 0) de ese componente GUI.*

El constructor **PanelAutoContenido** (línea 68) utiliza los métodos **addMouseListener** y **addMouseMotionListener** para registrar objetos de la clase interna anónima, para manejar los eventos del ratón y los eventos de movimiento del ratón para el **PanelAutoContenido**. En realidad, sólo se redefinen los métodos **mousePressed** (línea 72), **mouseReleased** (línea 78) y **mouseDragged** (línea 89) para realizar tareas. Los otros métodos manejadores de eventos de ratón se heredan de las clases **MouseAdapter** y **MouseMotionAdapter**, cuando se definen las clases internas anónimas.

Al extender la clase **JPanel**, en realidad creamos un nuevo componente GUI. A menudo, los administradores de diseño utilizan el método ***getPreferredSize*** de un componente GUI (heredado de la clase **java.awt.Component**) para determinar los mejores valores para el ancho y la altura de dicho componente cuando éste se diseña como parte de una GUI. Si un nuevo componente tiene un mejor ancho y altura, éste debe redefinir al método **getPreferredSize** (líneas 99 a 102) para devolver ese ancho y esa altura como objetos de la clase ***Dimension*** (del paquete **java.awt**).

### Observación de apariencia visual 29.18

*El tamaño predeterminado de un objeto **JPanel** es de 0 pixeles de ancho y de 0 pixeles de alto.*

### Observación de apariencia visual 29.19

*Al crear subclases de **JPanel** (o de cualquier otro **JComponent**), se debe redefinir el método **getPreferredSize** si el nuevo componente debe tener mejores valores para el ancho y la altura.*

El método **paintComponent** (línea 104) se redefine en la clase **PanelAutoContenido** para dibujar un óvalo. Para determinar el ancho, la altura y la esquina superior izquierda, el usuario debe oprimir el botón del ratón y mantenerlo así, y arrastrarlo y soltarlo en el área de dibujo del **PanelAutoContenido**.

Las coordenadas iniciales **x1** y **y1** en el área de dibujo del **PanelAutoContenido** se capturan en el método **mousePressed** (línea 72). A medida que el usuario arrastra el ratón después de la operación inicial en **mousePressed**, el programa genera una serie de llamadas a **mouseDragged** (línea 89) mientras el usuario continúa oprimiendo el botón del ratón y moviéndolo. Cada llamada captura en las variables **x2** y **y2** la posición actual del ratón con respecto a la esquina superior izquierda del **PanelAutoContenido**, y se hace una llamada a **repaint** para dibujar la versión actual del óvalo. La acción de dibujar queda confinada estrictamente al **PanelAutoContenido**, incluso si el usuario arrastra el ratón fuera del área de dibujo del **PanelAutoContenido**. Cualquier cosa que se dibuje fuera del **PanelAutoContenido** se *recorta*; los pixeles no se despliegan fuera de los límites del **PanelAutoContenido**.

Los cálculos proporcionados en el método **paintComponent** determinan la esquina superior izquierda apropiada utilizando dos veces el método **Math.min**, para encontrar el valor más pequeño para las coordenadas *x* y *y*. El ancho y la altura del óvalo deben ser valores positivos, pues de lo contrario éste no aparecerá en pantalla. El método **Math.abs** obtiene el valor absoluto de las restas **x1 – x2** y **y1 – y2** que determinan el ancho y la altura del rectángulo delimitador del óvalo, respectivamente. Cuando se completan los cálculos, **paintComponent** dibuja el óvalo. La llamada a la versión de **paintComponent** correspondiente a la superclase al principio del método garantiza que se borre el óvalo anterior mostrado en el **PanelAutoContenido**, antes de que el nuevo se despliegue en la pantalla.

### Observación de apariencia visual 29.20



*La mayoría de los componentes Swing pueden ser transparentes u opacos. Si un componente GUI de Swing es opaco, al llamar a su método **paintComponent** su fondo se borrará; en caso contrario, no se borrará.*

### Observación de apariencia visual 29.21



*La clase **JComponent** proporciona el método **setOpaque** que toma un argumento **booleano** para determinar si un objeto **JComponent** es opaco (**true**) o transparente (**false**).*

### Observación de apariencia visual 29.22



*Los objetos **JPanel** son opacos de manera predeterminada.*

Cuando el usuario suelta el botón del ratón, el método **`mouseReleased`** (línea 78) captura en las variables **`x1`** y **`y1`** la posición final del ratón e invoca al método **`repaint`** para dibujar la versión final del óvalo.

Cuando ejecute este programa, intente arrastrar el ratón desde el fondo de la ventana de la aplicación hacia el área del **`PanelAutoContenido`** para que vea que los eventos de arrastre se envían a la ventana de la aplicación, en lugar de enviarse al **`PanelAutoContenido`**. Después, inicie una nueva operación de arrastre en el área del **`PanelAutoContenido`** y arrastre el ratón hacia el fondo de la ventana de la aplicación, para que vea que los eventos de arrastre se envían al **`PanelAutoContenido`**, en lugar de enviarse a la ventana de la aplicación.

**Observación de apariencia visual 29.23**

*Una operación de arrastre de ratón empieza con un evento de oprimir el botón del ratón (**`mousePressed`**). Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a **`mouseDragged`**) se envían al componente GUI que recibió el evento original del botón oprimido del ratón.*

## 29.14 Ventanas

Hasta este punto hemos visto muchas aplicaciones que han utilizado una subclase de **`JFrame`** como la GUI de la aplicación. En esta sección hablaremos sobre varias cuestiones importantes relacionadas con los objetos **`JFrame`**.

Un objeto **`JFrame`** es una *ventana* con una *barra de título* y un *borde*. La clase **`JFrame`** es una subclase de ***`java.awt.Frame`*** (que a su vez es una subclase de **`java.awt.Window`**). Como tal, **`JFrame`** es uno de los pocos componentes GUI de Swing que no se considera ligero. A diferencia de la mayoría de los componentes Swing, **`JFrame`** no está escrito completamente en Java. De hecho, si usted despliega una ventana desde un programa en Java, la ventana forma parte del conjunto de componentes GUI de la plataforma local; la ventana se verá igual que las otras ventanas que se desplieguen en esa plataforma. Cuando un programa en Java se ejecuta en una Macintosh y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las demás aplicaciones de Macintosh. Cuando un programa en Java se ejecuta en Microsoft Windows y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las otras aplicaciones de Microsoft Windows. Y, cuando un programa en Java se ejecuta en una plataforma Unix y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las otras aplicaciones Unix en esa plataforma.

La clase **`JFrame`** soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, una ventana se oculta (es decir, desaparece de la pantalla) cuando el usuario la cierra. Esto puede controlarse mediante el método ***`setDefaultCloseOperation`*** de ***`JFrame`***. La interfaz ***`WindowConstants`*** (del paquete **`javax.swing`**) define tres constantes para usarse con este método: **`DISPOSE_ON_CLOSE`**, **`DO_NOTHING_ON_CLOSE`** y **`HIDE_ON_CLOSE`** (la opción predeterminada). La mayoría de las plataformas permiten desplegar un número limitado de ventanas en la pantalla. Como tal, una ventana es un recurso valioso que debe regresarse al sistema cuando ya no se necesita. La clase **`Window`** (una superclase indirecta de **`JFrame`**) define el método ***`dispose`*** para este propósito. Cuando un objeto **`Window`** ya no es necesario en una aplicación, usted debe usar **`dispose`** explícitamente para desechar la ventana. Esto puede hacerse mediante una llamada explícita al método **`dispose`** del objeto **`Window`**, o llamando al método **`setDefaultCloseOperation`** con el argumento **`WindowConstants.DISPOSE_ON_CLOSE`**. Además, al terminar una aplicación se regresarán los recursos de ventanas al sistema. Al establecer la operación predeterminada de cierre en **`DO_NOTHING_ON_CLOSE`**, usted estará indicando que determinará lo que debe hacerse cuando el usuario indique que la ventana debe cerrarse.



**Observación de ingeniería de software 29.4**

*Las ventanas son un recurso valioso del sistema, por lo que deben regresársele cuando ya no se les necesite.*

De manera predeterminada, una ventana no se despliega en la pantalla sino hasta que se llama a su método ***`show`***. Una ventana también puede mostrarse, llamando a su método ***`setVisible`*** (heredado de la clase **`java.awt.Component`**), con **`true`** como argumento. Además, el tamaño de una ventana debe establecerse mediante una llamada al método ***`setSize`*** (heredado de la clase **`java.awt.Component`**). La posición de una ventana al aparecer en la pantalla se especifica con el método **`setLocation`** (heredado de la clase **`java.awt.Component`**).

**Error común de programación 29.8**

*Olvidar llamar al método `show` o al método `setVisible` en una ventana, es un error lógico en tiempo de ejecución; la ventana no se desplegará en pantalla.*



**Error común de programación 29.9**

*Olvidar llamar al método `setSize` en una ventana, es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.*

Todas las ventanas generan *eventos de ventana* cuando el usuario las manipula. Los componentes que escuchan eventos se registran para los eventos de ventana por medio del método ***`addWindowListener`*** de **`Window`**. La interfaz ***`WindowListener`*** (implementada por los componentes de eventos de ventana) proporciona siete métodos para manejar los eventos de ventana: ***`windowActivated`*** (se llama cuando la ventana se activa al hacer clic en ella), ***`windowClosed`*** (se llama después de cerrar la ventana), ***`windowClosing`*** (se llama cuando el usuario inicia la operación de cierre de ventana), ***`windowDeactivated`*** (se llama cuando otra ventana se activa), ***`windowIconified`*** (se llama cuando el usuario minimiza una ventana), ***`windowDeiconified`*** (se llama cuando se restaura una ventana después de ser minimizada) y ***`windowOpened`*** (se llama cuando se despliega por primera vez una ventana en la pantalla).

La mayoría de las ventanas tienen un icono en la esquina superior izquierda o derecha, el cual permite al usuario cerrar la ventana y terminar el programa. La mayoría de las ventanas tienen también un icono en la esquina superior izquierda de la ventana, el cual despliega un menú cuando el usuario hace clic en el icono. Este menú por lo general contiene una opción **Cerrar** para cerrar la ventana y otras opciones para manipularla.

## 29.15 Uso de menús con marcos

Los *menús* son una parte integral de las GUIs. Los menús permiten al usuario realizar acciones sin "atestar" innecesariamente una interfaz gráfica de usuario con componentes GUI adicionales. En las GUIs de Swing, los menús pueden adjuntarse solamente a objetos de las clases que proporcionan el método **`setJMenuBar`**. Dos de esas clases son **`JFrame`** y **`JApplet`**. Las clases que se utilizan para definir menús son ***`JMenuBar`***, ***`JMenuItem`***, ***`JMenu`***, ***`JCheckBoxMenuItem`*** y la clase ***`JRadioButtonMenuItem`***.

**Observación de apariencia visual 29.24**




*Los menús simplifican las GUIs, al reducir el número de componentes que ve el usuario.*

La clase **`JMenuBar`** (una subclase de **`JComponent`**) contiene los métodos necesarios para administrar una *barra de menús*, la cual es un contenedor de menús.

La clase **`JMenuItem`** (una subclase de **`javax.swing.AbstractButton`**) contiene los métodos necesarios para administrar *elementos de menú*. Un elemento de menú es un componente GUI que se encuentra en un menú que, al ser seleccionado, hace que se realice una acción. Un elemento de menú puede usarse para iniciar una acción, o puede ser un *submenú* que proporcione más elementos de menú que pueda seleccionar el usuario. Los submenús son útiles para agrupar en un menú varios elementos de menú relacionados.

La clase **`JMenu`** (una subclase de **`javax.swing.JMenuItem`**) contiene los métodos necesarios para administrar *menús*. Los menús contienen elementos de menú y se agregan a las barras de menús o a otros menús como submenús. Al hacer clic en un menú, éste se expande para mostrar su lista de elementos. Al hacer clic en uno de los elementos del menú se genera un evento de acción.

La clase **`JCheckBoxMenuItem`** (una subclase de **`javax.swing.JMenuItem`**) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse. Cuando se selecciona un objeto **`JCheckBoxMenuItem`**, aparece una marca de verificación a la izquierda de ese elemento de menú. Cuando el objeto **`JCheckBoxMenuItem`** se selecciona nuevamente, se quita la marca de verificación que está a la izquierda del elemento de menú.

La clase **`JRadioButtonMenuItem`** (una subclase de **`javax.swing.JMenuItem`**) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse de igual forma que los objetos **`JCheckBoxMenuItem`**. Cuando se mantienen varios objetos **`JRadioButtonMenuItem`** como parte de un grupo de botones (**`ButtonGroup`**), sólo puede seleccionarse un elemento del grupo a la vez. Cuando se selecciona un objeto **`JRadioButtonMenuItem`**, aparece un círculo relleno a la izquierda del ele-

mento de menú. Cuando se selecciona otro objeto **JRadioButtonMenuItem**, se quita el círculo relleno que está a la izquierda del elemento de menú previamente seleccionado.

La aplicación de la figura 29.22 muestra el uso de varios tipos de elementos de menú. El programa también muestra cómo especificar caracteres especiales (conocidos como mnemónicos) que pueden proporcionar un acceso rápido a un menú o elemento de menú desde el teclado. Los mnemónicos pueden utilizarse con objetos de cualquier clase que sea subclase de **java.swing.AbstractButton**.

```
// Figura 29.22: PruebaMenu.java
// Demostración del uso de menús
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class PruebaMenu extends JFrame {
   private Color valoresColores[] =
      { Color.black, Color.blue, Color.red, Color.green };
   private JRadioButtonMenuItem elementosColores[], tiposDeLetra[];
   private JCheckBoxMenuItem elementosEstilo[];
   private JLabel pantalla;
   private ButtonGroup grupoTiposDeLetra, grupoColores;
   private int estilo;

   public PruebaMenu()
   {
      super( "Uso de objetos JMenu" );

      JMenuBar barra = new JMenuBar();  // crea la barra de menús
      setJMenuBar( barra );  // establece la barra de menús para el objeto
                                  JFrame

      // crea el menú Archivo y el elemento de menú Salir
      JMenu menuArchivo = new JMenu( "Archivo" );
      menuArchivo.setMnemonic( 'A' );
      JMenuItem elementoAcercaDe = new JMenuItem( "Acerca de..." );
      elementoAcercaDe.setMnemonic( 'c' );
      elementoAcercaDe.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               JOptionPane.showMessageDialog( PruebaMenu.this,
                  "Este es un ejemplo\ndel uso de menus",
                  "Acerca de", JOptionPane.PLAIN_MESSAGE );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      menuArchivo.add( elementoAcercaDe );

      JMenuItem elementoSalir = new JMenuItem( "Salir" );
      elementoSalir.setMnemonic( 'S' );
      elementoSalir.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               System.exit( 0 );
```

**Figura 29.22** Uso de objetos **JMenu** y mnemónicos; **PruebaMenu.java**. (Parte 1 de 4.)

```
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de addActionListener
      menuArchivo.add( elementoSalir );
      barra.add( menuArchivo );    // agrega el menú Archivo

      // crea el menú Formato, sus submenús y los elementos de menú
      JMenu menuFormato = new JMenu( “Formato” );
      menuFormato.setMnemonic( ‘F’ );

      // crea el submenú Color
      String colores[] =
         { “Negro”, “Azul”, “Rojo”, “Verde” };
      JMenu menuColor = new JMenu( “Color” );
      menuColor.setMnemonic( ‘C’ );
      elementosColores = new JRadioButtonMenuItem[ colores.length ];
      grupoColores = new ButtonGroup();
      ManejadorElementos manejadorElementos = new ManejadorElementos();

      for ( int i = 0; i < colores.length; i++ ) {
         elementosColores[ i ] =
            new JRadioButtonMenuItem( colores[ i ] );
         menuColor.add( elementosColores[ i ] );
         grupoColores.add( elementosColores[ i ] );
         elementosColores[ i ].addActionListener( manejadorElementos );
      } // fin de for

      elementosColores[ 0 ].setSelected( true );
      menuFormato.add( menuColor );
      menuFormato.addSeparator();

      // crea el submenú TipoDeLetra
      String nombresTiposDeLetra[] =
         { “TimesRoman”, “Courier”, “Helvetica” };
      JMenu menuTipoDeLetra = new JMenu( “Fuente” );
      menuTipoDeLetra.setMnemonic( ‘T’ );
      tiposDeLetra = new JRadioButtonMenuItem[ nombresTiposDeLetra.length ];
      grupoTiposDeLetra = new ButtonGroup();

      for ( int i = 0; i < tiposDeLetra.length; i++ ) {
         tiposDeLetra[ i ] =
            new JRadioButtonMenuItem( nombresTiposDeLetra[ i ] );
         menuTipoDeLetra.add( tiposDeLetra[ i ] );
         grupoTiposDeLetra.add( tiposDeLetra[ i ] );
         tiposDeLetra[ i ].addActionListener( manejadorElementos );
      } // fin de for

      tiposDeLetra[ 0 ].setSelected( true );
      menuTipoDeLetra.addSeparator();

      String nombresEstilos[] = { “Negrita”, “Cursiva” };
      elementosEstilo = new JCheckBoxMenuItem[ nombresEstilos.length ];
      ManejadorEstilos manejadorEstilos = new ManejadorEstilos();

```

**Figura 29.22** Uso de objetos `JMenu` y mnemónicos; `PruebaMenu.java`. (Parte 2 de 4.)

```
      for ( int i = 0; i < nombresEstilos.length; i++ ) {
         elementosEstilo[ i ] =
            new JCheckBoxMenuItem( nombresEstilos[ i ] );
         menuTipoDeLetra.add( elementosEstilo[ i ] );
         elementosEstilo[ i ].addItemListener( manejadorEstilos );
      } // fin de for

      menuFormato.add( menuTipoDeLetra );
      barra.add( menuFormato );  // agrega el menú Formato

      pantalla = new JLabel(
         "Texto muestra", SwingConstants.CENTER );
      pantalla.setForeground( valoresColores[ 0 ] );
      pantalla.setFont(
         new Font( "TimesRoman", Font.PLAIN, 72 ) );

      getContentPane().setBackground( Color.cyan );
      getContentPane().add( pantalla, BorderLayout.CENTER );

      setSize( 500, 200 );
      show();
   } // fin del constructor PruebaMenu

   public static void main( String args[] )
   {
      PruebaMenu ap = new PruebaMenu();

      ap.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
         {
            System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de addWindowListener
   } // fin de main

   class ManejadorElementos implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         for ( int i = 0; i < elementosColores.length; i++ )
            if ( elementosColores[ i ].isSelected() ) {
               pantalla.setForeground( valoresColores[ i ] );
               break;
            }

         for ( int i = 0; i < tiposDeLetra.length; i++ )
            if ( e.getSource() == tiposDeLetra[ i ] ) {
               pantalla.setFont( new Font(
                  tiposDeLetra[ i ].getText(), estilo, 72 ) );
               break;
            }

         repaint();
```

**Figura 29.22** Uso de objetos **`JMenu`** y mnemónicos; **`PruebaMenu.java`**. (Parte 3 de 4.)

```
155          } // fin del método actionPerformed
156       } // fin de la clase interna ManejadorElementos
157
158       class ManejadorEstilos implements ItemListener {
159          public void itemStateChanged( ItemEvent e )
160          {
161             estilo = 0;
162
163             if ( elementosEstilo[ 0 ].isSelected() )
164                estilo += Font.BOLD;
165
166             if ( elementosEstilo[ 1 ].isSelected() )
167                estilo += Font.ITALIC;
168
169             pantalla.setFont( new Font(
170                pantalla.getFont().getName(), estilo, 72 ) );
171
172             repaint();
173          } // fin del método itemStateChanged
174       } // fin de la clase interna ManejadorEstilos
175    } // fin de la clase PruebaMenu
```

Menú
Caracteres mnemónicos
Uso de objetos JMenu
Archivo Formato
Texto muestra
Barra de menús

Elementos de menú
Uso de objetos JMenu
Archivo Formato
Color
Fuente
TimesRoman
Courier
Helvetica
Negrita
Cursiva
Submenú expandido
Barra separadora

**Figura 29.22** Uso de objetos **`JMenu`** y mnemónicos; **`PruebaMenu.java`**. (Parte 4 de 4.)

La clase **`PruebaMenu`** (línea 7) es una clase completamente autocontenida: define todos los componentes GUI y el manejo de eventos para los elementos de menú. La mayor parte del código para esta aplicación aparece en el constructor de la clase (línea 16).

Las líneas 20 y 21

```
JMenuBar barra = new JMenuBar();  // crea la barra de menús
setJMenuBar( barra );  // establece la barra de menús para el objeto JFrame
```

crean el objeto **`JMenuBar`** y se adjunta a la ventana de la aplicación mediante el método ***`setJMenuBar`*** de **`JFrame`**.



### Error común de programación 29.10

*Olvidar establecer la barra de menús con el método **`setJMenuBar`** de **`JFrame`** hará que la barra de menús no se despliegue en el objeto **`JFrame`**.*

Las líneas 24 a 51 establecen el menú **Archivo** y se adjunta a la barra de menús. Este menú contiene un elemento de menú llamado **Acerca de...**, el cual muestra un cuadro de diálogo de mensaje cuando se selecciona, y un elemento de menú llamado **Salir** que puede seleccionarse para terminar la aplicación.

La línea 24

```
JMenu menuArchivo = new JMenu( "Archivo" );
```

crea un objeto **JMenu**, se asigna a la referencia **menuArchivo**, y se pasa al constructor la cadena **"Archivo"** como el nombre del menú. La línea 25

```
menuArchivo.setMnemonic( 'A' );
```

utiliza el método ***setMnemonic*** de **AbstractButton** (heredado a la clase **JMenu**) para indicar que **A** es el *mnemónico* de este menú. Al oprimir la tecla *Alt* y la letra ***A*** se abre el menú, exactamente igual que al hacer clic en el nombre del menú con el ratón. En la GUI, el carácter mnemónico del nombre del menú aparece subrayado (vea las capturas de pantalla).



**Observación de apariencia visual 29.25**

*Los mnemónicos proporcionan un acceso rápido con el teclado a los comandos de menú y de botón.*



**Observación de apariencia visual 29.26**

*Deben usarse distintos mnemónicos para cada botón o elemento de menú. En general, se utiliza la primera letra de la etiqueta correspondiente al elemento de menú o al botón como mnemónico. Si varios botones o elementos de menú empiezan con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo, la letra* **u** *se utiliza comúnmente para un botón o elemento de menú llamado* **Guardar como...**).

Las líneas 26 y 27

```
JMenuItem elementoAcercaDe = new JMenuItem( "Acerca de..." );
elementoAcercaDe.setMnemonic( 'c' );
```

definen el objeto **elementoAcercaDe** de **JMenuItem** con el nombre **"Acerca de..."** y establecen su mnemónico como la letra **'c'**. Este elemento de menú se agrega a **menuArchivo** en la línea 38. Para acceder al elemento **Acerca de...** por medio del teclado, oprima la tecla *Alt* y la letra *A* para abrir el menú **Archivo** y después oprima *c* para seleccionar el elemento de menú **Acerca de...**. Las líneas 28 a 37 crean un objeto **ActionListener** para escuchar la selección de **elementoAcercaDe**. Las líneas 32 a 34

```
JOptionPane.showMessageDialog( PruebaMenu.this,
   "Este es un ejemplo\ndel uso de menus",
   "Acerca de", JOptionPane.PLAIN_MESSAGE );
```

muestran un cuadro de diálogo de mensaje. En la mayoría de las veces que utilizamos **showMessageDialog**, el primer argumento fue **null**. El propósito del primer argumento es especificar la *ventana padre* para el cuadro de diálogo. Esta ventana padre ayuda a determinar en dónde se va a desplegar el cuadro de diálogo. Si la ventana padre se especifica como **null**, el cuadro de diálogo se despliega en el centro de la pantalla. Si la ventana padre no es **null**, el cuadro de diálogo se despliega centrado horizontalmente sobre la ventana padre, y justo debajo de la parte superior de la ventana.

Los cuadros de diálogo pueden ser *modales* o *no modales*. Un *cuadro de diálogo modal* no permite el acceso a ninguna otra ventana de la aplicación, sino hasta que el cuadro de diálogo se cierra. Un *cuadro de diálogo no modal* permite el acceso a otras ventanas mientras éste se despliega en pantalla. De manera predeterminada, los cuadros de diálogo desplegados con la clase **JoptionPane** son cuadros de diálogo modales. Usted puede usar la clase ***Jdialog*** para crear sus propios cuadros de diálogo modales o no modales.

La línea 38

```
menuArchivo.add( elementoAcercaDe );
```

agrega **elementoAcercaDe** al **menuArchivo** mediante el método ***add*** de **JMenu**.

Las líneas 40 a 50 definen el elemento de menú **elementoSalir**, establecen su mnemónico como **S** y registran un objeto **ActionListener** que termina la aplicación cuando se selecciona **elementoSalir**.

La línea 51

```
barra.add( menuArchivo );    // agrega el menú Archivo
```

utiliza el método ***add*** de **`JMenuBar`** para adjuntar el **`menuArchivo`** a **`barra`**.




**Observación de apariencia visual 29.27**

*Los menús normalmente aparecen de izquierda a derecha, en el orden en el que se agregan.*

Las líneas 54 y 55 crean el menú **`menuFormato`** y establecen su mnemónico como **`F`**.

Las líneas 60 y 61 crean el menú **`menuColor`** (éste será un submenú del menú **Formato**) y establecen su mnemónico como **`C`**. La línea 62 crea el arreglo **`JRadioButtonMenuItem`** llamado **`elementosColores`**, el cual hará referencia a los elementos de menú que se encuentran en **`menuColor`**. La línea 63 crea el objeto **`ButtonGroup`** llamado **`grupoColores`**, el cual se asegurará de que solamente se seleccione uno de los elementos de menú del submenú **Color** en un momento dado. La línea 64 define una instancia de la clase interna **`ManejadorElementos`** (definida en las líneas 138 a 156), la cual se utilizará para responder a las selecciones de los submenús **Color** y **Fuente** (que describiremos en breve). La estructura **`for`** de las líneas 66 a 72 crea cada objeto **`JRadioButtonMenuItem`** en el arreglo **`elementosColores`**, agrega cada elemento de menú a **`menuColor`**, agrega cada elemento de menú a **`grupoColores`** y registra el objeto **`ActionListener`** para cada elemento de menú.

La línea 74

```
elementosColores[ 0 ].setSelected( true );
```

utiliza el método **`setSelected`** de **`AbstractButton`** para indicar que el primer elemento del arreglo **`elementosColores`** debe estar seleccionado. La línea 75 agrega el **`menuColor`** como un submenú del **`menuFormato`**.




**Observación de apariencia visual 29.28**

*Agregar un menú como elemento de otro menú lo convierte automáticamente en un submenú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el mnemónico de ese submenú), éste se expande para mostrar sus elementos.*

La línea 76

```
menuFormato.addSeparator();
```

agrega una línea *separadora* al menú. El separador aparece como una línea horizontal en el menú.




**Observación de apariencia visual 29.29**

*Es posible agregar separadores a un menú para agrupar los elementos en forma lógica.*

Las líneas 79 a 94 crean el submenú **Fuente** y varios objetos **`JRadioButtonMenuItem`**, e indican que el primer elemento del arreglo de objetos **`JRadioButtonMenuItem`** llamado **`tiposDeLetra`** debe estar seleccionado. La línea 98 crea un arreglo de objetos **`JCheckBoxMenuItem`** para representar los elementos de menú para especificar los estilos negrita y cursiva para la fuente. La línea 99 define una instancia de la clase interna **`ManejadorEstilos`** (definida en las líneas 158 a 174) para responder a los eventos de **`JCheckBoxMenuItem`**. La estructura **`for`** de las líneas 101 a 106 crea cada objeto **`JCheckBoxMenuItem`**, agrega cada elemento de menú a **`menuTipoDeLetra`** y registra el objeto **`ItemListener`** para cada elemento de menú. La línea 108 agrega **`menuTipoDeLetra`** como un submenú de **`menuFormato`**. La línea 109 agrega el **`menuFormato`** a **`barra`**.

Las líneas 111 a 115 crean un objeto **`JLabel`** en el que la fuente, el color y el estilo se controlan a través del menú **Formato**. El color inicial de primer plano se establece como el primer elemento del arreglo **`valoresColores`** (**`Color.black`**) y el tipo de letra inicial se establece como **`TimesRoman`** con estilo **`PLAIN`** y tamaño de 72 puntos. La línea 117 establece el color de fondo del panel de contenido de la ventana como **`Color.cyan`**, y la línea 118 adjunta el objeto **`JLabel`** a la región **`CENTER`** del diseño **`BorderLayout`** del panel de contenido.

El método **actiionPerformed** de la clase **ManejadorElementos** (línea 138) utiliza dos estructuras **for** para determinar cuál elemento de menú fuente o color generó el evento, y establece la fuente o el color del objeto **pantalla** de **JLabel**, respectivamente. La condición **if** de la línea 142 utiliza el método ***isSelected*** de **AbstractButton** para determinar cuál objeto **JRadioButtonMenuItem** para seleccionar colores está seleccionado. La condición **if** de la línea 148 utiliza el método ***getSource*** de **EventSource** para obtener una referencia al objeto **JRadioButtonMenuItem** que generó el evento. La línea 150 utiliza el método ***getText*** de **AbstractButton** para obtener el nombre del tipo de letra, del elemento de menú.

El método **itemStateChanged** de la clase **ManejadorEstilos** (línea 158) se llama si el usuario selecciona un objeto **JCheckBoxMenuItem** en el **menuTipoDeLetra**. Las líneas 163 y 166 determinan si uno o ambos objetos **JCheckBoxMenuItem** están seleccionados, y utiliza su estado combinado para determinar el nuevo estilo de la fuente.

**Observación de apariencia visual 29.30**

*Cualquier componente GUI ligero (es decir, un componente que sea subclase de* ***JComponent****) puede agregarse a un objeto* **JMenu** *o* **JMenuBar**.

## RESUMEN

- Una interfaz gráfica de usuario (GUI) presenta una interfaz ilustrada de un programa. Una GUI proporciona a un programa una “apariencia visual” única.
- Al proporcionar a distintas aplicaciones un conjunto consistente de componentes intuitivos de la interfaz del usuario, las GUIs permiten al usuario pasar más tiempo utilizando el programa de una manera más productiva.
- Las GUIs se crean a partir de componentes GUI (algunas veces conocidos como controles o “widgets”). Un componente GUI es un objeto visual con el que el usuario interactúa mediante el ratón o el teclado.
- Los componentes GUI de Swing están definidos en el paquete **javax.swing**. Los componentes Swing están escritos, se manipulan y se despliegan completamente en Java.
- Los componentes GUI originales del paquete **java.awt** del Abstract Windowing Toolkit están enlazados directamente con las herramientas de la interfaz gráfica de usuario de la plataforma local.
- Los componentes de Swing son componentes ligeros. Los componentes del AWT están enlazados a la plataforma local y algunas veces se les conoce como componentes pesados: dependen del sistema de ventanas de la plataforma local para determinar su funcionalidad y su apariencia visual.
- Varios componentes GUI de Swing son componentes GUI pesados: en especial, las subclases de **java.awt.Window** (como **JFrame**) que muestran ventanas en la pantalla. Los componentes GUI pesados de Swing son menos flexibles que los componentes ligeros.
- La mayor parte de las herramientas de cada componente GUI de Swing se hereda de las clases **Component**, **Container** y **JComponent** (la superclase para la mayoría de los componentes Swing).
- Un objeto **Container** es un área en la que pueden colocarse componentes.
- Los objetos **JLabel** proporcionan instrucciones o información textual en una GUI.
- El método **setToolTipText** de **JComponent** especifica la información de herramienta que se despliega siempre que el usuario posiciona el cursor del ratón sobre un objeto **JComponent** en la GUI.
- Muchos componentes Swing pueden mostrar imágenes especificando un objeto **Icon** como argumento para su constructor, o utilizando un método **setIcon**.
- La clase **ImageIcon** (del paquete **javax.swing**) soporta dos formatos de imagen: el Formato de intercambio de gráficos (GIF) y el Grupo unido de expertos en fotografía (JPEG).
- La interfaz **SwingConstants** (del paquete **javax.swing**) define un conjunto de constantes enteras comunes (como **SwingConstants.LEFT**) que se utilizan con muchos componentes Swing.
- De manera predeterminada, el texto de un objeto **JComponent** aparece a la derecha de la imagen, cuando el objeto **JComponent** contiene tanto texto como una imagen.
- Las alineaciones horizontal y vertical de un objeto **JLabel** pueden establecerse mediante los métodos **setHorizontalAlignment** y **setVerticalAlignment**. El método **setText** establece el texto que se despliega en la etiqueta. El método **getText** recupera el texto actual que aparece en una etiqueta. Los métodos **setHorizontalTextPosition** y **setVerticalTextPosition** especifican la posición del texto en una etiqueta.

- El método **setIcon** de **JComponent** establece el objeto **Icon** que va a mostrarse en un objeto **JComponent**. El método **getIcon** recupera el objeto **Icon** actual mostrado en un objeto **JComponent**.
- Las GUIs generan eventos cuando el usuario interactúa con ellas. La información acerca de un evento GUI se almacena en un objeto de una clase que extienda a **AWTEvent**.
- Para procesar un evento, el programador debe registrar un componente que escuche eventos, e implementar uno o más manejadores de eventos.
- Al uso de componentes de escucha en el manejo de eventos se le conoce como modelo de delegación de eventos: el procesamiento de un evento se delega a un objeto específico en el programa.
- Cuando ocurre un evento, el componente GUI con el que interactuó el usuario notifica a sus componentes de escucha registrados mediante una llamada al método manejador de eventos apropiado para cada componente de escucha.
- Los objetos **JTextField** y **JPasswordField** son áreas de una sola línea en las que el usuario puede introducir texto desde el teclado, o simplemente pueden mostrar texto. Un objeto **JPasswordField** muestra que se escribió un carácter a medida que el usuario va escribiendo, pero oculta automáticamente los caracteres.
- Cuando el usuario escribe datos en un objeto **JTextField** o **JPasswordField** y oprime *Entrar*, se genera un evento **ActionEvent**.
- El método **setEditable** de **JTextComponent** determina si el usuario puede modificar el texto de un objeto **JTextComponent**.
- El método **getPassword** de **JPasswordField** devuelve la contraseña como un arreglo de tipo **char**.
- Cada objeto **JComponent** contiene un objeto de la clase **EventListenerList** (del paquete **javax.swing.event**) llamado **listenerList**, en el que se almacenan todos los componentes de escucha registrados.
- Cada objeto **JComponent** sporta varios tipos de eventos distintos, que incluyen eventos de ratón, de tecla y otros más. Cuando ocurre un evento, éste se despacha (se envía) solamente a los componentes de escucha de eventos del tipo apropiado. Cada tipo de evento tiene su interfaz de escucha de eventos correspondiente.
- Cuando se genera un evento debido a la interacción de un usuario con un componente, al componente se le otorga un ID de evento único para especificar el tipo de evento. El componente GUI utiliza el ID de evento para decidir el tipo de componente de escucha al que debe despacharse el evento, junto con el método manejador de eventos al que debe llamar.
- Un objeto **JButton** genera un evento **ActionEvent** cuando el usuario hace clic en el botón con el ratón.
- Un objeto **AbstractButton** puede tener un objeto **Icon** de sustitución que se despliega cuando el ratón se coloca sobre el botón. El icono cambia a medida que el ratón se desplaza hacia adentro y hacia afuera del área del botón en la pantalla. El método **setRollOverIcon** de **AbstractButton** especifica la imagen a desplegar en un botón, cuando el usuario coloca el ratón sobre ese botón.
- Los componentes GUI de Swing contienen tres tipos de botones de estado (**JtoggleButton**, **JCheckBox** y **JRadioButton**) con valores de encendido/apagado o verdadero/falso. Las clases **JCheckBox** y **JRadioButton** son subclases de **JtoggleButton**.
- Cuando el usuario hace clic en un objeto **JCheckBox** se genera un evento **ItemEvent**, el cual puede ser manejado por un objeto **ItemListener**. Estos objetos deben definir al método **itemStateChanged**. El método **getStateChange** de **ItemEvent** determina el estado de un objeto **JToggleButton**.
- Los objetos **JRadioButton** son similares a los objetos **JCheckBox** en cuanto a que tienen dos estados: seleccionado y no seleccionado. Los objetos **JRadioButton** generalmente aparecen como un grupo en el que sólo puede haber un botón de opción seleccionado a la vez.
- Un objeto **JComboBox** (al que algunas veces se le conoce como lista desplegable) proporciona una lista de elementos para que el usuario seleccione uno de ellos. Los objetos **JComboBox** generan eventos **ItemEvent**. Un índice numérico lleva el registro del orden de los elementos en un objeto **JComboBox**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento agregado a un objeto **JComboBox** aparece como el elemento actualmente seleccionado cuando se despliega el objeto **JComboBox**. El método **getSelectedIndex** de **JComboBox** devuelve el número del índice correspondiente al elemento seleccionado.
- Es posible atrapar eventos de ratón para cualquier componente GUI que se derive de **java.awt.Component** por medio de objetos **MouseListener** y **MouseMotionListener**.
- Cada método manejador de eventos de ratón toma como su argumento un objeto **MouseEvent** que contiene información acerca del evento de ratón y la ubicación en donde ocurrió el evento.
- Los métodos **addMouseListener** y **addMouseMotionListener** son métodos de **Component** utilizados para registrar componentes que escuchan eventos de ratón para un objeto de cualquier clase que extienda a **Component**.

- Muchas de las interfaces que escuchan eventos proporcionan varios métodos. Para cada uno de ellos hay su correspondiente clase adaptadora de escucha de eventos, la cual proporciona una implementación detallada de cada método en la interfaz. El programador puede extender la clase adaptadora para heredar la implementación predeterminada de cada método y simplemente redefinir el método o métodos necesarios para el manejo de eventos en el programa.
- El método **getClickCount** de **MouseEvent** devuelve el número de clics del ratón.
- Los métodos **isMetaDown** e **isAltDown** de **InputEvent** se utilizan para determinar en qué botón hizo clic el usuario.
- Los administradores de diseños ordenan los componentes GUI en un contenedor para fines de presentación.
- **FlowLayout** distribuye los componentes de izquierda a derecha, en el orden en el que se agregan al contenedor. Al llegar al borde del contenedor, los componentes continúan en la siguiente línea.
- El método **setAlignment** de **FlowLayout** cambia la alineación del diseño **FlowLayout** a **FlowLayout.LEFT**, **FlowLayout.CENTER** o **FlowLayout_RIGHT**.
- El administrador de diseño **BorderLayout** ordena los componentes en cinco regiones: Norte, Sur, Este, Oeste y Centro. Puede agregarse un componente a cada región.
- El método **layoutContainer** de **LayoutManager** recalcula la distribución de su argumento **Container**.
- El administrador de diseño **GridLayout** divide el contenedor en una cuadrícula de filas y columnas. Los componentes se agregan a un diseño **GridLayout**, empezando en la celda superior izquierda y procediendo de izquierda a derecha, hasta que la fila esté llena. Después, el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, etcétera.
- El método **validate** de **Container** recalcula la distribución del contenedor con base en el administrador de diseño actual para el objeto **Container** y el conjunto actual de componentes GUI desplegados en pantalla.
- Los paneles se crean mediante la clase **JPanel**, la cual hereda de la clase **JComponent**. Se pueden agregar componentes a los objetos **JPanel**, incluso otros paneles.
- Los objetos **JTextArea** proporcionan un área para manipular varias líneas de texto. Al igual que la clase **JTextField**, la clase **JTextArea** hereda de **JTextComponent**.
- Un evento externo (es decir, un evento generado por un componente GUI distinto) generalmente indica cúando debe procesarse el texto en un objeto **JTextArea**.
- Para un objeto **JTextArea** se proporcionan barras de desplazamiento, si éste se adjunta a un objeto **JScrollPane**.
- El método **getSelectedText** devuelve el texto seleccionado de un objeto **JTextArea**. El texto se selecciona arrastrando el ratón sobre el texto deseado para resaltarlo.
- El método **setText** establece el texto en un objeto **JTextArea**.
- Para proporcionar la envoltura automática de palabras en un objeto **JTextArea**, adjúntelo a un objeto **JScrollPane** con la directiva de barra de desplazamiento horizontal **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**.
- Las directivas de barras de desplazamiento horizontal y vertical para un objeto **JScrollPane** se establecen cuando se crea, o por medio de los métodos **setHorizontalScrollBarPolicy** y **setVerticalScrollBarPolicy** de la clase **JScrollPane**.
- Un objeto **JPanel** puede utilizarse como área dedicada de dibujo, la cual puede recibir eventos de ratón y a menudo se extiende para crear nuevos componentes GUI.
- Los componentes Swing que heredan de la clase **JComponent** contienen el método **paintComponent**, el cual los ayuda a dibujar adecuadamente dentro del contexto de una GUI de Swing. El método **paintComponent** de **JComponent** debe redefinirse de la siguiente manera:

```
public void paintComponent( Graphics g )
{
   super.paintComponent( g );

   //  el código adicional de dibujo
}
```

- La llamada a la versión de **paintComponent** correspondiente a la superclase garantiza que la acción de dibujar ocurra en el orden adecuado y que el mecanismo de dibujo de Swing permanezca intacto. Si no se hace una llamada a la versión de **paintComponent** correspondiente a la superclase, por lo general el componente GUI personalizado no se desplegará apropiadamente en la interfaz de usuario. Además, si se hace la llamada a la versión de la superclase después de ejecutar las instrucciones de dibujo personalizadas, los resultados por lo general se borran.

- Las clases **JFrame** y **JApplet** no son subclases de **JComponent**; por lo tanto, no contienen el método **paintComponent** (tienen el método **paint**).
- Al llamar a **repaint** para un componente GUI de Swing se indica que el componente deberá dibujarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. De manera predeterminada, la mayoría de los componentes Swing son transparentes. El método **setOpaque** de **JComponent** puede recibir un argumento **booleano** que indica si el componente es opaco (**true)** o transparente (**false**). Los componentes GUI del paquete **java.awt** son distintos de los componentes Swing, en cuanto a que **repaint** produce una llamada al método **update** de **Component** (el cual borra el fondo del componente) y **update** a su vez llama al método **paint** (en lugar de llamar a **paintComponent**).
- El método **setTitle** despliega un objeto **String** en la barra de título de una ventana.
- Para dibujar en cualquier componente GUI se requieren coordenadas que se miden a partir de la esquina superior izquierda *(0, 0)* de ese componente GUI.
- Los administradores de diseño a menudo utilizan el método **getPreferredSize** de un componente GUI para determinar los mejores valores para el ancho y la altura, al distribuir ese componente como parte de una GUI. Si un nuevo componente tiene un mejor valor de ancho y altura, debe redefinirse el método **getPreferredSize** para que devuelva ese ancho y esa altura como un objeto de la clase **Dimension** (del paquete **java.awt**).
- El tamaño predeterminado de un objeto **JPanel** es de 0 pixeles de ancho y 0 pixeles de alto.
- Una operación de arrastre de ratón empieza con un evento de botón oprimido del ratón. Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a **mouseDragged**) se envían al componente GUI que recibió el evento original de botón oprimido del ratón.
- Un objeto **JFrame** es una ventana con una barra de título y un borde. La clase **JFrame** es una subclase de **java.awt.Frame** (que a su vez es una subclase de **java.awt.Window**).
- La clase **JFrame** soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, cuando el usuario cierra una ventana, el objeto **JFrame** se oculta. Esto puede controlarse mediante el método **setDefaultCloseOperation** de **JFrame**. La interfaz **WindowConstants** (del paquete **javax.swing**) define tres constantes para usarse con este método: **DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE** y **HIDE_ON_CLOSE** (la opción predeterminada).
- De manera predeterminada, una ventana no se despliega en la pantalla sino hasta que se llama a su método **show**. Una ventana también puede desplegarse llamando a su método **setVisible**, con **true** como argumento.
- El tamaño de una ventana debe establecerse mediante una llamada al método **setSize**. La posición que tendrá una ventana al aparecer en pantalla se especifica mediante el método **setLocation**.
- Todas las ventanas generan eventos de ventana cuando el usuario las manipula. Los componentes que escuchan eventos se registran para los eventos de ventana por medio del método **addWindowListener** de la clase **Window**. La interfaz **WindowListener** proporciona siete métodos para manejar eventos de ventana: **windowActivated** (se llama cuando la ventana se convierte en la ventana activa al hacer clic en ella), **windowClosed** (se llama después de cerrar la ventana), **windowClosing** (se llama cuando el usuario inicia el proceso de cerrar la ventana), **windowDeactivated** (se llama cuando otra ventana se convierte en la ventana activa), **windowIconified** (se llama cuando el usuario minimiza una ventana), **windowDeiconified** (se llama cuando una ventana se restaura, después de estar minimizada) y **windowOpened** (se llama cuando se muestra una ventana por primera vez en la pantalla).
- Los argumentos de línea de comandos se pasan automáticamente a **main** como el arreglo de objetos **String** llamado **args**. El primer argumento después del nombre de la clase de la aplicación es el primer objeto **String** del arreglo **args**, y la longitud del arreglo es el número total de argumentos de la línea de comandos.
- Los menús son una parte integral de las GUIs, ya que permiten al usuario realizar acciones sin "atestar" innecesariamente una interfaz gráfica de usuario con componentes GUI adicionales.
- En las GUIs de Swing, los menús sólo pueden adjuntarse a objetos de las clases que proporcionan el método **setJMenuBar**. Dos de esas clases son **JFrame** y **JApplet**.
- Las clases utilizadas para definir menús son **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** y **JRadioButtonMenuItem**.
- Un objeto **JMenuBar** es un contenedor de menús.
- Un objeto **JMenuItem** es un componente GUI dentro de un menú que, cuando se selecciona, hace que se lleve a cabo cierta acción. Un objeto **JMenuItem** puede usarse para iniciar una acción o puede ser un submenú que proporcione más elementos de menú que el usuario pueda seleccionar.

- Un objeto **JMenu** contiene elementos de menú y puede agregarse a un objeto **JMenuBar** o a otros objetos **JMenu** como submenú. Al hacer clic en un menú, éste se expande para mostrar su lista de elementos.
- Al seleccionar un objeto **JCheckBoxMenuItem** aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona nuevamente este objeto **JCheckBoxMenuItem**, la marca de verificación desaparece.
- Cuando se mantienen varios objetos **JRadioButtonMenuItem** como parte de un objeto **ButtonGroup**, sólo puede seleccionarse un elemento del grupo a la vez. Cuando se selecciona un objeto **JRadioButtonMenuItem**, aparece un círculo relleno a la izquierda del elemento de menú. Cuando se selecciona otro **JRadioButtonMenuItem**, se quita el círculo relleno a la izquierda del elemento de menú previamente seleccionado.
- El método **setJMenuBar** de **JFrame** adjunta una barra de menús a un objeto **JFrame**.
- El método **setMnemonic** de **AbstractButton** (heredado en la clase **JMenu**) especifica el mnemónico para un objeto **AbstractButton**. Al oprimir la tecla *Alt* y el mnemónico se lleva a cabo la acción del objeto **AbstractButton** (en el caso de un menú, éste se abre).
- Los caracteres mnemónicos normalmente aparecen subrayados.
- Los cuadros de diálogo pueden ser modales o no modales. Un cuadro de diálogo modal no permite el acceso a ninguna otra ventana en la aplicación, sino hasta que el cuadro de diálogo se cierra. Un cuadro de diálogo no modal permite el acceso a otras ventanas mientras se despliega en pantalla. De manera predeterminada, los cuadros de diálogo que se despliegan mediante la clase **JOptionPane** son cuadros de diálogo modales. La clase **JDialog** puede usarse para crear cuadros de diálogo modales o no modales.
- El método **addSeparator** de **JMenu** agrega una línea separadora a un menú.

## TERMINOLOGÍA

Abstract Windows Toolkit
administrador de diseño
administrador de diseño **BoxLayout**
alineado a la derecha
alineado a la izquierda
área de dibujo dedicada
arrastrar
barra de desplazamiento
barra de herramientas
barra de menús
**BorderLayout.CENTER**
**BorderLayout.EAST**
**BorderLayout.NORTH**
**BorderLayout.SOUTH**
**BorderLayout.WEST**
botón
botón de comando
botón de opción
casilla de verificación
clase **AbstractButton**
clase **ActionEvent**
clase adaptadora
clase **BorderLayout**
clase **Box**
clase **Component**
clase **ComponentAdapter**
clase **Container**
clase **ContainerAdapter**
clase **EventListenerList**
clase **EventObject**
clase **FlowLayout**
clase **FocusAdapter**
clase **GridLayout**
clase **ImageIcon**
clase **ItemEvent**
clase **JButton**
clase **JCheckBox**
clase **JCheckBoxMenuItem**
clase **JComboBox**
clase **JComponent**
clase **JLabel**
clase **JMenu**
clase **JMenuBar**
clase **JMenuItem**
clase **JPanel**
clase **JPasswordField**
clase **JRadioButton**
clase **JScrollPane**
clase **JTextArea**
clase **JTextComponent**
clase **JTextField**
clase **JToggleButton**
clase **MouseAdapter**
clase **MouseEvent**
clase **MouseMotionAdapter**
clase **WindowAdapter**
componente GUI
componente GUI de Swing
componente ligero
componente pesado
componente que escucha eventos
control
controlado por eventos
cuadro de desplazamiento
cuadro de diálogo modal
cuadro de diálogo no modal
despachar un evento
directivas de barra de desplazamiento para un objeto **JScrollPane**
elemento de menú
envoltura automática de palabras
"escuchar" un evento
espacio libre horizontal
espacio libre vertical
etiqueta
etiqueta de botón
etiqueta de casilla de verificación
evento
evento externo
extensión **.gif** de nombre de archivo
extensión **.jpg** de nombre de archivo
flecha de desplazamiento
**FlowLayout.CENTER**
**FlowLayout.LEFT**
**FlowLayout.RIGHT**
**Font.BOLD**
**Font.ITALIC**
**Font.PLAIN**
Formato de intercambio de gráficos (GIF)
Grupo unido de expertos en fotografía (JPEG)
icono de sustitución
ID de evento
información de herramientas

interfaz **ActionListener**
interfaz **ComponentListener**
interfaz **ContainerListener**
interfaz **FocusListener**
interfaz **Icon**
interfaz **ItemListener**
interfaz **LayoutManager**
interfaz **MouseListener**
interfaz **MouseMotionListener**
interfaz que escucha eventos
interfaz **SwingConstants**
interfaz **windowListener**
**ItemEvent.DESELECTED**
**ItemEvent.SELECTED**
justificado a la izquierda
lista de selección múltiple
lista de selección simple
lista desplegable
localización de la interfaz de usuario
manejador de eventos
menú
método **actionPerformed**
método **add** de la clase **Container**
método **addItemListener**
método **addMouseListener**
método **addMouseMotion-Listener**
método **addSeparator** de la clase **JMenu**
método **addWindowListener** de **Window**
método **dispose** de la clase **Window**
método **getActionCommand**
método **getIcon**
método **getMinimumSize** de **Component**
método **getPassword** de **JPasswordField**
método **getPreferredSize** de **Component**
método **getSelectedIndex** de **JComboBox**
método **getSelectedIndex** de **JList**
método **getSelectedText**
método **getSelectedValues** de **JList**
método **getSource** de **ActionEvent**
método **getStateChange** de **ItemEvent**
método **getText** de **JLabel**
método **getX** de **MouseEvent**
método **getY** de **MouseEvent**
método **itemStateChanged**
método **layoutContainer**
método **mouseClicked**
método **mouseDragged**
método **mouseEntered**
método **mouseExited**
método **mouseMoved**
método **mousePressed**
método **mouseReleased**
método **paintComponent** de **JComponent**
método **setAlignment**
método **setBackground**
método **setDefaultClose-Operation**
método **setEditable**
método **setHorizontal-ScrollBarPolicy**
método **setIcon**
método **setJMenuBar**
método **setLayout** de la clase **Container**
método **setListData** de **JList**
método **setMaximumRowCount**
método **setMnemonic** de **AbstractButton**
método **setOpaque** de la clase **JComponent**
método **setRollOverIcon**
método **setSelected** de **AbstractButton**
método **setSelectionMode**
método **setText**
método **setTitle** de la clase **Frame**
método **setToolTipText**
método **setVertical-Alignment**
método **setVerticalScroll-BarPolicy**
método **setVerticalText-Position**
método **setVisible**
método **setVisibleRowCount**
método **validate**
método **valueChanged**
método **windowActivated**
método **windowClosed**
método **windowClosing**
método **windowDeactivated**
método **windowDeiconified**
método **windowIconified**
método **windowOpened**
mnemónico
modelo de delegación de eventos
modo de selección
paquete **java.awt**
paquete **java.awt.event**
paquete **javax.swing**
paquete **javax.swing.event**
registrar un componente que escucha eventos
sistema de ventanas
submenú
**SwingConstants.HORIZONTAL**
**SwingConstants.VERTICAL**
tecla de método abreviado (mnemónicos)
texto de sólo lectura
ventana
widgets o controles (accesorios de ventana)
**windowConstants.DISPOSE_ON_CLOSE**

## *ERRORES COMUNES DE PROGRAMACIÓN*

**29.1** Olvidar agregar un componente a un contenedor, para que pueda mostrarse en pantalla, es un error lógico en tiempo de ejecución.

**29.2** Si se agrega a un contenedor un componente que no se haya instanciado, se lanza una excepción **NullPointer-Exception**.

**29.3** Utilizar una letra **f** minúscula en los nombres de las clases **JTextField** o **JPasswordField**, es un error de sintaxis.

**29.4** Olvidar registrar un objeto manejador de eventos para un tipo de evento de un componente GUI en particular, da como resultado que no se manejen los eventos de ese componente.

**29.5** Si se agrega más de un componente a una región específica en un diseño **BorderLayout**, sólo se desplegará el último componente que se haya agregado. No hay un mensaje de error para indicar este problema.

**29.6** Cuando se redefine el método **paintComponent** de un objeto **JComponent**, si no se hace una llamada a la versión original de **paintComponent** de la superclase, el componente GUI no podrá desplegarse apropiadamente en la GUI.

**29.7** Cuando se redefine el método **paintComponent** de un objeto **JComponent**, al llamar a la versión original de **paintComponent** de la superclase después de realizar otro dibujo, se borran los demás dibujos.

**29.8** Olvidar llamar al método **show** o al método **setVisible** en una ventana, es un error lógico en tiempo de ejecución; la ventana no se desplegará en pantalla.

**29.9** Olvidar llamar al método **setSize** en una ventana, es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.

**29.10** Olvidar establecer la barra de menús con el método **setJMenuBar** de **JFrame** hará que la barra de menús no se despliegue en el objeto **JFrame**.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**29.1** Estudie los métodos de la clase **Component** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de la mayoría de los componentes GUI.

**29.2** Estudie los métodos de la clase **Container** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.

**29.3** Estudie los métodos de la clase **JComponent** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.

**29.4** Estudie los métodos de la clase **javax.swing.JLabel** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas completas de la clase antes de usarla.

**29.5** Utilice clases separadas para procesar eventos GUI.

## OBSERVACIONES DE APARIENCIA VISUAL

**29.1** Las interfaces de usuario consistentes permiten a un usuario aprender a utilizar nuevas aplicaciones en menos tiempo.

**29.2** Los componentes Swing están escritos en Java, por lo que ofrecen un mayor nivel de portabilidad y flexibilidad que los componentes GUI originales de Java del paquete **java.awt**.

**29.3** Utilice los cuadros de información de herramienta (establecidos mediante el método **setToolTipText** de **JComponent**) para agregar texto descriptivo a sus componentes GUI. Este texto ayuda al usuario a determinar el propósito del componente GUI en la interfaz de usuario.

**29.4** A menudo, un evento externo determina cuándo debe procesarse el texto de un objeto **JTextArea**.

**29.5** Para proporcionar la funcionalidad de envoltura automática de palabras para un objeto **JTextArea**, invoque al método **setLineWrap** con un argumento **true**.

**29.6** Tener más de un objeto **JButton** con la misma etiqueta hace que los objetos **JButton** sean ambiguos para el usuario. Asegúrese de proporcionar una etiqueta única para cada botón.

**29.7** El uso de iconos de sustitución para objetos **JButton** proporciona al usuario una retroalimentación visual, la cual le indica que, si hace clic en el ratón, se realizará la acción del botón.

**29.8** La clase **AbstractButton** soporta que se despliegue texto e imágenes en un botón, por lo que todas las subclases de **AbstractButton** también soportan el despliegue de texto e imágenes.

**29.9** Establezca el conteo máximo de filas para un objeto **JComboBox** en un número que evite que la lista se expanda más allá de los límites de la ventana o del applet en que se utilice. Esto garantizará que la lista aparezca correctamente cuando el usuario la expanda.

**29.10** Las llamadas al método **mouseDragged** se envían al objeto **MouseMotionListener** para el objeto **Component** en el que se inició la operación de arrastre. De manera similar, la llamada al método **mouseReleased** se envía al objeto **MouseListener** para el objeto **Component** en el que se inició la operación de arrastre.

**29.11** La mayoría de los entornos de programación de Java proporcionan herramientas de diseño GUI, las cuales ayudan a un programador a diseñar de manera gráfica una GUI, y después escriben automáticamente el código de Java necesario para crear la GUI.

**29.12** Cada contenedor puede tener solamente un administrador de diseño a la vez (varios contenedores en el mismo programa pueden tener distintos administradores de diseño).

**29.13** Si no se especifica una región al agregar un objeto **Component** a un diseño **BorderLayout**, se asume que el objeto **Component** va a agregarse a la región **BorderLayout.CENTER**.

**29.14** Combinar gráficos y componentes GUI puede ocasionar un despliegue incorrecto de los gráficos, de los componentes GUI o de ambos. Utilizar objetos **JPanel** para dibujar puede eliminar este problema, proporcionando un área de dibujo dedicada para los gráficos.

**29.15** Cuando se redefine el método **paintComponent** de un objeto **JComponent**, la primera instrucción del cuerpo siempre debe ser una llamada a la versión original del método de la superclase.

**29.16** Llamar a **repaint** para un componente GUI de Swing indica que ese componente debe pintarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. La mayoría de los componentes Swing son transparentes de manera predeterminada. Es posible pasar un argumento booleano al método **setOpaque** de **JComponent** para indicar si el componente es opaco (**true**), o transparente (**false**). Los componentes GUI del paquete **java.awt** son distintos de los componentes Swing en cuanto a que **repaint** produce una llamada al método **update** de **Component** (con lo cual se borra el fondo del componente), y **update**, a su vez, llama al método **paint** (en lugar de llamar a **paintComponent**).

**29.17** El proceso de dibujar en cualquier componente GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0, 0) de ese componente GUI.

**29.18** El tamaño predeterminado de un objeto **JPanel** es de 0 pixeles de ancho y de 0 pixeles de alto.

**29.19** Al crear subclases de **JPanel** (o de cualquier otro **JComponent**), se debe redefinir el método **getPreferredSize** si el nuevo componente debe tener mejores valores para el ancho y la altura.

**29.20** La mayoría de los componentes Swing pueden ser transparentes u opacos. Si un componente GUI de Swing es opaco, al llamar a su método **paintComponent** su fondo se borrará; en caso contrario, no se borrará.

**29.21** La clase **JComponent** proporciona el método **setOpaque** que toma un argumento **booleano** para determinar si un objeto **JComponent** es opaco (**true**) o transparente (**false**).

**29.22** Los objetos **JPanel** son opacos de manera predeterminada.

**29.23** Una operación de arrastre de ratón empieza con un evento de oprimir el botón del ratón (**mousePressed**). Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a **mouseDragged**) se envían al componente GUI que recibió el evento original del botón oprimido del ratón.

**29.24** Los menús simplifican las GUIs, al reducir el número de componentes que ve el usuario.

**29.25** Los mnemónicos proporcionan un acceso rápido con el teclado a los comandos de menú y de botón.

**29.26** Deben usarse distintos mnemónicos para cada botón o elemento de menú. En general, se utiliza la primera letra de la etiqueta correspondiente al elemento de menú o al botón como mnemónico. Si varios botones o elementos de menú empiezan con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo, la letra **u** se utiliza comúnmente para un botón o elemento de menú llamado **Guardar como...**).

**29.27** Los menús normalmente aparecen de izquierda a derecha, en el orden en el que se agregan.

**29.28** Agregar un menú como elemento de otro menú lo convierte automáticamente en un submenú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el mnemónico de ese submenú), éste se expande para mostrar sus elementos.

**29.29** Es posible agregar separadores a un menú para agrupar los elementos en forma lógica.

**29.30** Cualquier componente GUI ligero (es decir, un componente que sea subclase de **JComponent**) puede agregarse a un objeto **JMenu** o **JMenuBar**.

## *TIP DE PORTABILIDAD*

**29.1** La apariencia de una GUI definida con componentes GUI pesados del paquete **java.awt** puede variar entre plataformas. Los componentes pesados se "enlazan" a la GUI de la plataforma "local", la cual varía entre las distintas plataformas.

## *OBSERVACIONES DE INGENIERÍA DE SOFTWARE*

**29.1** Para utilizar componentes GUI con efectividad debe comprender las jerarquías de herencia de **javax.swing** y **java.awt**; en especial de las clases **Component**, **Container** y **JComponent**, que definen características comunes para la mayoría de los componentes Swing.

**29.2** El componente que escucha un evento dado deberá implementar la interfaz para escuchar eventos apropiada.

**29.3** Utilizar clases separadas para manejar eventos GUI produce componentes de software más reutilizables, confiables y legibles, los cuales pueden colocarse en paquetes y utilizarse en muchos programas.

**29.4** Las ventanas son un recurso valioso del sistema, por lo que deben regresársele cuando ya no se les necesite.

## *EJERCICIOS DE AUTOEVALUACIÓN*

**29.1** Complete los espacios en blanco:

a) El método ____________ es llamado cuando el ratón se mueve y un componente que escucha eventos está registrado para manejar el evento.
b) El texto que no puede ser modificado por el usuario se llama texto ____________.
c) Un ____________ ordena los componentes GUI en un objeto **`Container`**.
d) El método **`add`** para adjuntar componentes GUI es un método de la clase ____________.
e) GUI es un acrónimo de ____________.
f) El método ____________ se utiliza para establecer el administrador de diseño para un contenedor.
g) Una llamada al método **`mouseDragged`** va después de una llamada al método ____________ y antes de una llamada al método ____________.

**29.2** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) **`BorderLayout`** es el administrador de diseño predeterminado para un panel de contenido.
b) Cuando el cursor del ratón se mueve hacia los límites de un componente GUI, se hace una llamada al método **`mouseOver`**.
c) Un objeto **`JPanel`** no puede agregarse a otro **`JPanel`**.
d) En un diseño **`BorderLayout`**, dos botones que se agreguen a la región **`NORTH`** aparecerán uno al lado del otro.
e) Cuando se utiliza **`BorderLayout`**, puede usarse un máximo de cinco componentes.

**29.3** Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).

a)
```
nombreBoton = JButton( “Leyenda” );
```
b)
```
JLabel unaEtiqueta, JLabel;     // crea referencias
```
c)
```
campoTexto = new JTextField( 50, “Texto predeterminado” );
```
d)
```
Container c = getContentPane();
setLayout( new BorderLayout() );
boton1 = new JButton( “Estrella del norte” );
boton2 = new JButton( “Polo sur” );
c.add( boton1 );
c.add( boton2 );
```

## *RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN*

**29.1** a) **`mouseMoved`**. b) No editable (de sólo lectura). c) Administrador de diseño. d) **`Container`**. e) Interfaz gráfica de usuario. f) **`setLayout`**. g) **`mousePressed`**, **`mouseReleased`**.

**29.2** a) Verdadero.
b) Falso. Se hace una llamada al método **`mouseEntered`**.
c) Falso. Un **`JPanel`** puede agregarse a otro **`JPanel`**, ya que **`JPanel`** es una subclase indirecta de **`Component`**. Por lo tanto, un **`JPanel`** es un **`Component`**. Cualquier **`Component`** puede agregarse a un **`Container`**.
d) Falso. Sólo se desplegará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región de un diseño **`BorderLayout`**.
e) Verdadero.

**29.3** a) se necesita **`new`** para crear un objeto.
b) **`JLabel`** es el nombre de una clase y no puede utilizarse como nombre de variable.
c) Los argumentos que se pasan al constructor están invertidos. El objeto **`String`** debe pasarse primero.
d) Se ha establecido **`BorderLayout`** y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones **`add`** apropiadas serían:
```
contenedor.add( boton1, BorderLayout.NORTH );
contenedor.add( boton2, BorderLayout.SOUTH );
```

## EJERCICIOS

**29.4** Complete los espacios en blanco:

a) La clase **`JTextField`** hereda directamente de __________.

b) Los administradores de diseño que describimos en este capítulo son __________, __________ y __________.

c) El método __________ de **`Container`** adjunta un componente GUI a un contenedor.

d) El método __________ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).

**29.5** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Sólo puede usarse un administrador de diseño por cada objeto **`Container`**.

b) En un diseño **`BorderLayout`**, los componentes GUI pueden agregarse a un **`Container`** en cualquier orden.

c) El método **`setFont`** de **`Graphics`** se utiliza para establecer la fuente de los campos de texto.

d) Un objeto **`Mouse`** contiene un método llamado **`mouseDragged`**.

**29.6** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Un objeto **`JApplet`** no tiene panel de contenido.

b) Un objeto **`JPanel`** es un objeto **`JComponent`**.

c) Un objeto **`JPanel`** es un objeto **`Component`**.

d) Un objeto **`JLabel`** es un objeto **`Container`**.

e) Un objeto **`AbstractButton`** es un objeto **`JButton`**.

g) Un objeto **`JTextField`** es un objeto **`Object`**.

**29.7** Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.

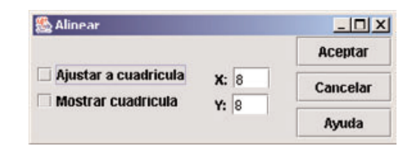
a) **`import javax.swing.*     // incluye el paquete swing`**

b) **`objetoPanel.GridLayout( 8, 8 ); // establece el diseño GridLayout`**

c) **`c.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`**

d) **`c.add( botonEste, EAST );// BorderLayout`**

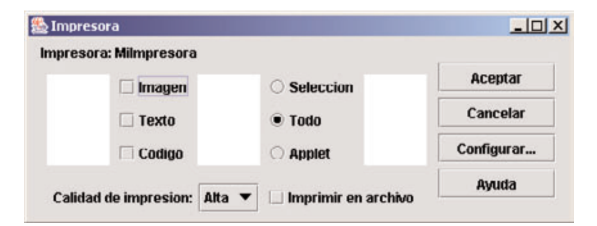
**29.8** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.




**29.9** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.




**29.10** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.




**29.11** Escriba un programa de conversión de temperatura, que convierta grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto **`JTextField`**). Debe usarse un objeto **`JLabel`** para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

***`Centígrados = 5/9 × ( Fahrenheit – 32 )`***

**29.12** Escriba una aplicación que permita al usuario dibujar un rectángulo, arrastrando el ratón en la ventana de aplicación. La coordenada superior izquierda deberá ser la ubicación en donde el usuario oprima el botón del ratón, y la coordenada inferior derecha deberá ser la ubicación en donde el usuario suelte el botón del ratón. Además, muestre el área del rectángulo en un **`JLabel`**, en la región **`SOUTH`** de un diseño **`BorderLayout`**. Todo el proceso de dibujo deberá realizarse en una subclase de **`JPanel`**. Use la siguiente fórmula para el área:

***`área = ancho x altura`***

**29.13** Escriba un programa que muestre un círculo de tamaño aleatorio, que calcule y muestre el área, el radio, el diámetro y la circunferencia. Use las siguientes ecuaciones: *diámetro* = $2 \times$ *radio, área* = $\pi \times$ *radio*$^2$, *circunferencia* = $2 \times \pi \times$ *radio.* Use la constante **`Math.PI`** para pi ($\pi$). Todos los dibujos deberán realizarse en una subclase de **`JPanel`** y los resultados de los cálculos deberán mostrarse en un objeto **`JTextArea`** de sólo lectura.

**29.14** Escriba un programa que utilice instrucciones **`System.out.println`** para imprimir los eventos según ocurran. Proporcione un objeto **`JComboBox`** con un mínimo de cuatro elementos. El usuario deberá ser capaz de seleccionar del objeto **`JComboBox`** un evento a "vigilar". Cuando ocurra ese evento específico, muestre información acerca de él en un cuadro de diálogo de mensaje. Use el método **`toString`** en el objeto evento para convertirlo en una representación de cadena.

**29.15** Escriba un programa utilizando métodos de la interfaz **`MouseListener`**, que permita al usuario oprimir el botón del ratón, arrastrar el ratón y soltar el botón del ratón. Cuando se suelte el botón del ratón, dibuje un rectángulo con la esquina superior izquierda, el ancho y la altura adecuados. [*Pista:* El método **`mousePressed`** debe capturar el conjunto de coordenadas en donde el usuario oprime inicialmente el botón del ratón y lo mantiene así, y el método **`mouseReleased`** debe capturar el conjunto de coordenadas en donde el usuario suelta el botón del ratón. Ambos métodos deberán almacenar los valores de coordenada apropiados. Todos los dibujos deberán realizarse en una subclase de **`JPanel`**, y todos los cálculos del ancho, la altura y la esquina superior izquierda deben realizarse mediante el método **`paintComponent`**, antes de que se dibuje la figura.]

**29.16** Modifique el ejercicio 29.15 para proporcionar un efecto de "banda de hule". Conforme el usuario arrastre el ratón, deberá poder ver el tamaño actual del rectángulo para saber exactamente cómo se verá el rectángulo cuando suelte el botón del ratón. [*Pista:* El método **`mouseDragged`** debe realizar las mismas tareas que **`mouseReleased`**.]

**29.17** Modifique el ejercicio 29.16 para permitir al usuario seleccionar cuál figura dibujar. Un objeto **`JComboBox`** debe proporcionar opciones que incluyan, cuando menos, rectángulo, óvalo, línea y rectángulo redondeado.

**29.18** Modifique el ejercicio 29.17 para permitir al usuario seleccionar el color de dibujo desde un cuadro de diálogo **`JColorChooser`**.

**29.19** Modifique el ejercicio 29.18 para permitir al usuario especificar si una figura debe llenarse o vaciarse cuando ésta se dibuja. El usuario deberá hacer clic en un objeto **`JCheckBox`** para indicar si está llena o vacía.

**29.20** (*Aplicación de dibujo completa.*) Por medio de las técnicas desarrolladas en los ejercicios 29.12 a 29.19, cree un programa de dibujo completo. Este programa debe utilizar los componentes GUI que vimos en este capítulo para permitir al usuario seleccionar la figura, el color y las características de relleno. Para este programa, cree sus propias clases (al igual que las de la jerarquía de clases que describimos en el ejercicio 27.19), a partir de las cuales se crearán objetos para guardar cada figura que dibuje el usuario. Las clases deberán almacenar la ubicación, las dimensiones y el color de cada figura, y deberán indicar si está llena o vacía. Sus clases deben derivarse de una clase llamada **`MiFigura`** que tenga todas las características comunes de cada tipo de figura. Cada subclase de **`MiFigura`** debe tener su propio método **`draw`**, el cual deberá devolver **`void`** y recibir un objeto **`Graphics`** como su argumento. Cree una subclase de **`JPanel`** llamada **`PanelDibujo`** para dibujar las figuras. Al llamar al método **`paintComponent`** de **`PanelDibujo`**, éste deberá recorrer el arreglo de figuras y mostrar cada una de ellas mediante una llamada polimórfica al método **`draw`** de la figura (con el objeto **`Graphics`** como argumento). El método **`draw`** de cada figura debe saber cómo dibujar la figura. Como mínimo, su programa debe proporcionar las siguientes clases: **`MiLinea`**, **`MiOvalo`**, **`MiRectangulo`**, **`MiRectanguloRedondeado`**. Diseñe la jerarquía de clases para obtener una máxima reutilización del código, y coloque todas sus clases en el paquete **`figuras`**. Importe este paquete en su programa. Cada figura debe almacenarse en un arreglo de objetos **`MiFigura`**, en donde **`MiFigura`** será la superclase en su jerarquía de clases de figuras (vea el ejercicio 27.19).

**29.21** Modifique el ejercicio 29.20 para proporcionar un botón **Deshacer** que pueda utilizarse varias veces para deshacer la última operación de dibujo. Si no hay figuras en el arreglo de figuras, el botón **Deshacer** debe estar deshabilitado.



# Multimedia en Java: Imágenes, animación y audio




## Objetivos

- Comprender cómo obtener y desplegar imágenes.
- Crear animaciones a partir de secuencias de imágenes; controlar la velocidad y el parpadeo de animación.
- Obtener, reproducir, repetir y detener sonidos.
- Dar seguimiento a la carga de imágenes con la clase **`MediaTracker`**; crear mapas de imágenes.
- Personalizar los **`applets`** con la etiqueta **`param`**.

*La llanta que más rechina al rodar es la que obtiene el aceite.*
John Billings (Henry Wheeler Shaw)

*El ruido no demuestra nada. Con frecuencia, una gallina que tan solo pone un huevo, cacarea como si hubiera puesto un asteroide.*
Mark Twain

*Utilizaremos una señal que ya he utilizado, que se reconoce a lo lejos y es fácil de gritar. ¡Waa-huuu!*
Zane Grey

*Una pantalla grande solamente hace que una mala película sea doblemente mala.*
Samuel Goldwyn

*Entre el movimiento y el acto existe la sombra.*
Thomas Stearns Eliot

*Lo que experimentamos de la naturaleza es con modelos, y todos los modelos de la naturaleza son muy hermosos.*
Richard Buckminster Fuller

## Plan general



## 30.1 Introducción

Bienvenido a lo que probablemente representa la mayor revolución en la historia de la industria de la computación. Aquellos de nosotros que entramos al gremio hace algunas décadas estábamos interesados primordialmente en el uso de las computadoras para hacer cálculos numéricos a gran velocidad. Pero conforme evoluciona el campo de las computadoras, comenzamos a darnos cuenta de que en la actualidad también es igualmente importante la manipulación de datos. La "chispa" de Java es la *multimedia*, el uso de *sonido*, *imagen*, *gráficos* y *vídeo* para hacer que las aplicaciones "cobren vida". En la actualidad, mucha gente considera al video en color de dos dimensiones como lo "último" en multimedia. Pero dentro de una década, esperamos toda clase de aplicaciones novedosas y excitantes en tres dimensiones. La programación multimedia ofrece muchos retos nuevos. El campo ya es enorme y crecerá rápidamente.

La gente se está apresurando para equipar a sus computadoras con multimedia. La mayoría de las computadoras nuevas se venden "listas para multimedia" con dispositivos de CD y DVD, tarjetas de sonido y, algunas veces, con capacidades especiales de vídeo.

Entre los usuarios que desean gráficos, los de dos dimensiones ya no son suficientes. Ahora mucha gente quiere gráficos en tres dimensiones, de alta resolución y en color. Las imágenes reales en tres dimensiones estarán disponibles a lo largo de la siguiente década. Imagine tener televisión de ultra alta resolución, con "teatro circular" y en tres dimensiones. Los eventos deportivos y de entretenimiento ¡tendrán lugar en su propia habitación! Los estudiantes de medicina alrededor del mundo verán las operaciones que se realizan a miles de kilómetros, como si ocurrieran en la misma habitación. La gente será capaz de aprender en sus casas a conducir por medio de simuladores extremadamente realistas, antes de colocarse frente al volante. Las posibilidades son excitantes e interminables.

La multimedia exige un extraordinario poder de cómputo. Hasta hace muy poco, las computadoras con este tipo de potencia no estaban disponibles. Pero los procesadores ultrarápidos actuales como el SPARC Ultra de Sun Microsystems, el Pentium de Intel, el Alpha de Compaq Computer Corporation y el R8000 de MIPS/Silicon Graphics (entre otros) están haciendo posible la multimedia. Las industrias de cómputo y de comunicaciones serán las principales beneficiarias de la revolución de la multimedia. Los usuarios estarán dispuestos a pagar por procesadores más rápidos, más memoria y anchos de banda más grandes que se necesitarán para soportar las aplicaciones multimedia. Irónicamente, es probable que los usuarios no tengan que pagar más, ya que la feroz competencia de estas industrias hace que los precios bajen.

Necesitamos lenguajes de programación para hacer más fácil la creación de aplicaciones multimedia. La mayoría de los lenguajes de programación no tienen incluidas las capacidades multimedia. Pero Java, a través de los paquetes de clases que son parte integral del mundo de la programación en Java, proporciona facilidades extendidas para multimedia que le permitirán comenzar a desarrollar de inmediato poderosas aplicaciones multimedia.

En este capítulo explicaremos una serie de ejemplos de "código vivo" que cubren muchas de las características multimedia que necesitará para construir aplicaciones útiles. Explicaremos los fundamentos de la manipula-

ción de imágenes, la creación de animaciones suaves, la reproducción de sonidos, la reproducción de vídeos, la creación de mapas de imágenes que pueden sentir cuando el apuntador se encuentra sobre ellos incluso sin un clic del ratón, y cómo personalizar los applets mediante los parámetros suministrados desde el archivo HTML que invoca al applet. Los ejercicios del capítulo sugieren proyectos interesantes y desafiantes, e incluso mencionan algunas ideas valiosas ¡que le podrían ayudar a hacer una fortuna! Cuando creamos estos ejercicios, las ideas seguían fluyendo. Con certeza, la multimedia promoverá la creatividad de formas que no hemos experimentado con las capacidades de cómputo "convencionales".

## 30.2 Cómo cargar, desplegar y escalar imágenes

Las capacidades multimedia de Java incluyen gráficos, imágenes, animaciones, sonidos y vídeo. Comenzaremos nuestra explicación de multimedia con las imágenes.

El applet de la figura 30.1 muestra cómo cargar una ***Imagen*** (**Image** del paquete **java.awt**) y cómo cargar un ***ImageIcon*** (del paquete **javax.swing**). El applet despliega la **Imagen** en su tamaño original y con una escala al doble de su longitud y altura original mediante dos versiones del método ***drawImage*** de **Graphics**. Además, el applet dibuja el **ImageIcon** mediante el método ***paintIcon***. La clase **Image-Icono** es particularmente útil debido a que se puede utilizar para cargar fácilmente una imagen dentro de un applet o una aplicación.

```
// Figura 30.1: CargaYEscalaDeImagen.java
// Carga una imagen y la despliega en su tamaño original
// y la escala al doble de su ancho y altura.
// Carga y despliega la misma imagen como un IconoImagen.
import java.applet.Applet;
import java.awt.*;
import javax.swing.*;

public class CargaYEscalaDeImagen extends JApplet {
    private Image logo1;
    private ImageIcon logo2;

    // carga la imagen cuando se carga el applet
    public void init()
    {
        logo1 = getImage( getDocumentBase(), "logo.gif" );
        logo2 = new ImageIcon( "logo.gif" );
    } // end method init

    // despliega la imagen
    public void paint( Graphics g )
    {
        // dibuja la imagen original
        g.drawImage( logo1, 0, 0, this );

        // dibuja la imagen escalada para que coincida con el ancho del applet
        // y con la altura del applet menos 120 pixeles
        g.drawImage( logo1, 0, 120,
                     getWidth(), getHeight() - 120, this );

        // dibuja el icono utilizando su método paintIcon
        logo2.paintIcon( this, g, 180, 0 );
    } // fin del método paint
} // fin de la clase CargaYEscalaDeImagen
```

**Figura 30.1** Cómo cargar y desplegar una imagen dentro de un applet. (Parte 1 de 2.)




**Figura 30.1** Cómo cargar y desplegar una imagen dentro de un applet. (Parte 2 de 2.)

Las líneas 10 y 11 declaran una referencia a **`Image`** y una referencia a **`ImageIcon`**, respectivamente. La clase **`Image`** es una clase **`abstract`**; por lo tanto, usted no puede crear un objeto directamente de la clase **`Image`**. En vez de eso, debe pedir que se cargue y se le devuelva una **`Image`**. La clase **`Applet`** (la superclase de **`JApplet`**) proporciona un método que hace precisamente eso. La línea 16 en el método **`init`** del applet

```
logo1 = getImage( getDocumentBase(), "logo.gif" );
```

utiliza el método ***`getImage`*** de **`Applet`** para cargar una **`Imagen`** dentro del applet. La versión de **`getImage`** toma dos argumentos, la ubicación en donde se almacena la imagen y el nombre del archivo de la imagen. En el primer argumento utilizamos el método **`getDocumentBase`** de **`Applet`** para determinar la ubicación de la imagen en Internet (o en su computadora si es de ahí de donde proviene). Asumimos que la imagen que se va a cargar se almacena en el mismo directorio que el archivo HTML que invoca al applet. El método **`getDocumentBase`** devuelve la ubicación del archivo HTML en Internet como un objeto de la clase ***`URL`*** (del paquete **`java.net`**). Una **`URL`** almacena un *Localizador Uniforme (o Universal) de Recursos*; un formato estándar para una dirección de una pieza de información en Internet. El segundo argumento especifica el nombre del archivo de la imagen. Actualmente Java soporta dos formatos de imagen, el *GIF* (*Formato de Intercambio de Gráficos)* y el *JPEG (Grupo unido de expertos en fotografía)*. Los nombres de archivos para cada tipo terminan con **`.gif`** o **`.jpg`** (o **`.jpeg`**) respectivamente.

### Tip de portabilidad 30.1

*La clase **`Image`** es una clase **`abstract`**, por lo que no pueden crearse objetos de **`Image`** de manera directa. Para lograr la independencia de la plataforma, la implementación de Java en cada plataforma proporciona su propia subclase de **`Image`** para almacenar la información de la imagen.*

Cuando se invoca al método **`getImage`**, se lanza un subproceso de ejecución separado en el que se carga la imagen (o se descarga desde Internet). Esto permite al programa continuar la ejecución mientras se carga la imagen. [*Nota:* Si el archivo requerido no está disponible, el método **`getImage`** no indica un error.]

La clase **`ImageIcon`** no es una clase **`abstract`**; por lo tanto, usted puede crear un objeto a partir de **`ImageIcon`**. La línea 17 del método **`init`** del applet,

```
logo2 = new ImageIcon( "logo.gif");
```

crea un objeto de **`ImageIcon`** que carga la misma imagen **`logo.gif`**. La clase **`ImageIcon`** proporciona muchos constructores que permiten inicializar con una imagen a un objeto **`ImageIcon`** desde la computadora local, o con una imagen almacenada en el servidor Web en Internet. La línea 24

```
g.drawImage( logo1, 0, 0, this );
```

utiliza el método **drawImage** de **Graphics**, el cual recibe cuatro argumentos (en realidad existen seis versiones sobrecargadas de este método). El primer argumento es una referencia al objeto **Image** en el que se almacena la imagen (**logo1**). El segundo y el tercer argumentos son las coordenadas *x* y *y* en donde debe desplegarse la imagen sobre el applet (las coordenadas indican la esquina superior izquierda de la imagen). El último argumento es una referencia a un objeto ***ImageObserver***. Por lo general, el **ImageObserver** es un objeto sobre el que se despliega la imagen; utilizamos **this** para indicar el applet. Un **imageObserver** puede ser cualquier objeto que implemente la interfaz **ImageObserver**. La interfaz **ImageObserver** se implementa mediante la clase **Component** (una de las superclases indirectas de **Applet**). De esta manera, todos los **Component** pueden ser **ImageObserver**. Este argumento es importante cuando se despliegan imágenes de gran tamaño que requieren mucho tiempo para descargarse desde Internet. Es posible que un programa despliegue la imagen antes de que se complete la descarga. Al **ImageObserver** se le notifica automáticamente para que actualice la imagen que se desplegó, mientras se carga el resto de la imagen. Cuando ejecute este applet, observe con atención cómo se despliegan las piezas de la imagen mientras ésta se carga. [*Nota:* En las computadoras más rápidas, podría no notarse este efecto.]

Las líneas 28 y 29

```
g.drawImage( logo1, 0, 120,
             getWidth(), getHeight() – 120, this );
```

utilizan otra versión del método **drawImage** de **Graphics** para desplegar una versión *a escala* de la imagen. El cuarto y quinto argumentos especifican la *longitud* y la *altura* de la imagen para propósitos del desplegado. La imagen se escala automáticamente para que coincida con la longitud y la altura especificadas. El cuarto argumento indica que la longitud de la imagen a escala debe ser la longitud del applet y el quinto argumento indica que la altura debe ser de 120 pixeles menor que la altura del applet. La longitud y la altura del applet se determinan con los métodos **getWidth** y **getHeight** (que se heredan de la clase **Component**).

La línea 32

```
logo2.paintIcon( this, g, 180, 0);
```

utiliza el método ***paintIcon*** de **ImageIcon** para desplegar la imagen. El método requiere cuatro argumentos, una referencia al **Componente** en el que se desplegará la imagen, una referencia al objeto, una referencia al objeto **Graphics** que se utilizará para modelar la imagen, y las coordenadas *x* y *y* de la esquina superior izquierda de la imagen.

Si compara las dos formas en las que cargamos y desplegamos las imágenes en este ejemplo, podrá ver que utilizar **ImageIcon** es más sencillo. Usted puede crear directamente objetos de la clase **ImageIcon** y no necesita utilizar una referencia a **ImageObserver** cuando despliega la imagen. Por esta razón, utilizaremos la clase **ImageIcon** en el resto del capítulo. [*Nota:* El método **paintIcon** de la clase **ImageIcon** no permite el escalamiento de una imagen. Sin embargo, la clase proporciona el método **getImage**, el cual devuelve una referencia a **Image** que puede utilizarse con el método **drawImage** de **Graphics** para desplegar la imagen seleccionada.]

## 30.3 Cómo cargar y reproducir clips de audio

Los programas en Java pueden manipular y reproducir *clips de audio*. Para los usuarios es fácil capturar sus propios clips de audio, y existe una gran variedad de clips que están disponibles en los productos de software y en Internet. Su sistema necesita estar equipado con el hardware para audio (bocinas y una tarjeta de sonido) para que sea capaz de reproducir clips de audio.

Java proporciona dos mecanismos para la reproducción de sonidos dentro de un applet, el método ***play*** de **Applet** y el método ***play*** de la *interfaz* ***AudioCLip***. Si usted quisiera reproducir un sonido una vez en un programa, el método **play** de **Applet** cargará el sonido y lo reproducirá una sola vez; el sonido se marca para el recolector de basura cuando termina la reproducción. El método **play** de **Applet** tiene dos formatos:

```
public void play( ubicación URL, Cadena nombreArchivoAudio );
public void play( URL URLaudio );
```

La primera versión carga el clip de audio almacenado en el archivo **nombreArchivoAudio** desde la **ubicación URL**, y reproduce el sonido. Por lo general, el primer argumento es una llamada al *método* **getDo-**

**cumentBase** o ***getCodeBase***. El método **getDocumentBase** indica la ubicación del archivo HTML que cargó al applet. El método **getCodeBase** indica la ubicación del archivo **.class** del applet. La segunda versión del método **play** toma una **URL** que contiene la ubicación y el nombre del archivo del clip de audio. La instrucción

```
play( gtDocumentBase(), “hola.au” );
```

carga el clip de audio en el archivo **hola.au** y lo reproduce una vez.

El *motor de audio* que reproduce los clips de audio soporta distintos formatos de archivos de sonido que incluyen el *formato de archivo de sonido de Sun* (*extensión* ***.au***), el *formato de archivo Wave de Windows* (*extensión* ***.wav***), el *formato de archivo AIFF de Macintosh* (*extensiones* ***.aif*** *o* ***.aiff***) y el *formato de archivo Musical Instrument Digital Interface* (*MIDI*) (*extensiones* **.mid** o **.rmi**).

La figura 30.2 muestra la carga y la reproducción de un ***AudioClip*** (del paquete **java.applet**). Esta técnica es más flexible que el método **play** de **Applet**, ya que permite almacenar el sonido en el programa, de manera que se pueda reutilizar a lo largo de la ejecución del programa. El método ***getAudioClip*** de **Audio** tiene dos formas que toman los mismos argumentos que el método **play** descrito anteriormente. El método **getAudioClip** devuelve una referencia a un **AudioClip**. Una vez que se carga **AudioClip,** se pueden invocar tres métodos para el objeto: ***play***, ***loop*** y ***stop***. El método **play** reproduce el sonido solamente una vez. El método **loop** ejecuta repetidamente un clip de audio de fondo. El método **stop** termina el clip de audio que se encuentra en reproducción. En el programa, cada uno de estos métodos se asocia con un botón del applet.

```
// Figura 30.2: CargaYReproduccionDeAudio.java
// Carga un clip de audio y lo reproduce.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CargaYReproduccionDeAudio extends JApplet {
   private AudioClip sonido1, sonido2, sonidoActual;
   private JButton reproduceSonido, repiteSonido, detieneSonido;
   private JComboBox eligeSonido;

   // carga la imagen cuando el applet comienza su ejecución
   public void init()
   {
      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      String elecciones[] = { “Bienvenido”, “Hola” };
      eligeSonido = new JComboBox( elecciones );
      eligeSonido.addItemListener(
         new ItemListener() {
            public void itemStateChanged( ItemEvent e )
            {
               sonidoActual.stop();

               sonidoActual =
                  eligeSonido.getSelectedIndex() == 0 ?
                     sonido1 : sonido2;
            } // fin del método itemStateChanged
         } // fin de la clase interna anónima
      ); // fin de addItemListener
```

**Figura 30.2** Cómo cargar y reproducir un **AudioClip**. (Parte 1 de 2.)

```
33         c.add( eligeSonido );
34
35         ButtonHandler manejador = new ButtonHandler();
36         reproduceSonido = new JButton( "Reproducir" );
37         reproduceSonido.addActionListener( manejador );
38         c.add( reproduceSonido );
39         repiteSonido = new JButton( "Repetir" );
40         repiteSonido.addActionListener( manejador );
41         c.add( repiteSonido );
42         detieneSonido = new JButton( "Detener" );
43         detieneSonido.addActionListener( manejador );
44         c.add( detieneSonido );
45
46         sonido1 = getAudioClip(
47                     getDocumentBase(), "bienvenido.wav" );
48         sonido2 = getAudioClip(
49                     getDocumentBase(), "hola.au" );
50         sonidoActual = sonido1;
51      } // fin del método init
52
53      // detiene el sonido cuando el usuario intercambia las páginas Web
54      // (es decir, sea amable con el usuario)
55      public void stop()
56      {
57         sonidoActual.stop();
58      } // fin del método stop
59
60      private class ButtonHandler implements ActionListener {
61         public void actionPerformed( ActionEvent e )
62         {
63            if ( e.getSource() == reproduceSonido )
64               sonidoActual.play();
65            else if ( e.getSource() == repiteSonido )
66               sonidoActual.loop();
67            else if ( e.getSource() == detieneSonido )
68               sonidoActual.stop();
69         } // fin del método actionPerformed
70      } // fin de la clase interna ButtonHandler
71   } // fin de la clase CargaYReproduccionDeAudio
```

**Figura 30.2** Cómo cargar y reproducir un **AudioClip**. (Parte 2 de 2.)

Las líneas 46 a 49 del método **init** del applet

```
sonido1 = getAudioClip(
            getDocumentBase(), "bienvenido.wav" );
sonido2 = getAudioClip(
            getDocumentBase(), "hola.au" );
```

utilizan **getAudioClip** para cargar dos archivos de sonido: un archivo Wave de Windows (**bienvenido.wav**) y un archivo de audio de Sun (**hola.au**). El usuario puede seleccionar el clip de sonido a reproducir

desde **JComboBox.chooseSound**. Observe que el método **stop** del applet se redefine en la línea 55. Cuando el usuario intercambia las páginas Web, se invoca al método **stop** del applet. Esta versión de **stop** garantiza que un sonido en reproducción se detenga. De lo contrario, el clip de sonido continuará ejecutándose como fondo. En realidad éste no es un problema, pero puede ser tedioso para el usuario si el clip de sonido se repite. El método **stop** se proporciona aquí como un detalle para el usuario.

**Buena práctica de programación 30.1**

*Cuando reproduzca sonidos en un applet o en una aplicación, proporcione un mecanismo para que el usuario pueda deshabilitar el sonido.*

## 30.4 Cómo animar una serie de imágenes

El siguiente ejemplo muestra la animación de series de imágenes almacenadas en un arreglo. La aplicación utiliza las mismas técnicas para cargar y desplegar **ImageIcons** que aparecen en la figura 30.1. En las ediciones previas de este texto, utilizamos una serie de ejemplos de animación para mostrar distintas técnicas para suavizar una animación. Una de las técnicas clave involucra el concepto llamado *gráficos con doble buffer*. Sin embargo, debido a que las nuevas características de los componentes GUI de Swing ya implementan las técnicas de suavización, simplemente nos podemos concentrar en el concepto de la animación.

La animación que presentamos en la figura 30.3 está diseñada como una subclase de **JPanel** (llamada **AnimadorLogo**), de modo que se puede adjuntar a una ventana de aplicación o posiblemente a un **JApplet**. Además, la clase **AnimadorLogo** define un método **main** (definido en la línea 71) para ejecutar la animación como una aplicación. El método **main** define una instancia de la clase **JFrame** y adjunta un objeto **AnimadorLogo** al **JFrame** para desplegar la animación.

```
1  // Figura 30.3: AnimadorLogo.java
2  // Animación de una serie de imágenes
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class AnimadorLogo extends JPanel
8                              implements ActionListener {
9     protected ImageIcon imagenes[];
10    protected int totalImagenes = 30,
11                  imagenActual = 0,
12                    retardoAnimacion = 50; // 50 milisegundos de retardo
13    protected Timer cronoAnimacion;
14
15    public AnimadorLogo()
16    {
17       setSize( getPreferredSize() );
18
19       imagenes = new ImageIcon[ totalImagenes ];
20
21       for ( int i = 0; i < imagenes.length; ++i )
22          imagenes[ i ] =
23             new ImageIcon( “imagenes/deitel” + i + “.gif” );
24
25       iniciaAnimacion();
26    } // fin del constructor AnimadorLogo
27
28    public void paintComponent( Graphics g )
29    {
```

**Figura 30.3** Animación de una serie de imágenes. (Parte 1 de 3.)

```
      super.paintComponent( g );

      if ( imagenes[ imagenActual ].getImageLoadStatus() ==
           MediaTracker.COMPLETE ) {
         imagenes[ imagenActual ].paintIcon( this, g, 0, 0 );
         imagenActual = ( imagenActual + 1 ) % totalImagenes;
      }
   } // fin del método paintComponent

   public void actionPerformed( ActionEvent e )
   {
      repaint();
   } // fin del método actionPerformed

   public void iniciaAnimacion()
   {
      if ( cronoAnimacion == null ) {
         imagenActual = 0;
         cronoAnimacion = new Timer( retardoAnimacion, this );
         cronoAnimacion.start();
      }
      else  // continúa desde la última imagen desplegada
         if ( ! cronoAnimacion.isRunning() )
            cronoAnimacion.restart();
   } // fin del método iniciaAnimacion

   public void terminaAnimacion()
   {
      cronoAnimacion.stop();
   } // fin del método terminaAnimacion

   public Dimension getMinimumSize()
   {
      return getPreferredSize();
   } // fin del método getMinimumSize

   public Dimension getPreferredSize()
   {
      return new Dimension( 160, 80 );
   } // fin del método getPreferredSize

   public static void main( String args[] )
   {
      AnimadorLogo anim = new AnimadorLogo();

      JFrame app = new JFrame( "Prueba Animacion" );
      app.getContentPane().add( anim, BorderLayout.CENTER );

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
```

**Figura 30.3** Animación de una serie de imágenes. (Parte 2 de 3.)

```
85          ); // fin del addWindowListener
86
87          // Las constantes 10 y 30 se utilizan abajo para establecer el tamaño de
88          // la ventana 10 pixeles más ancha que la animación y
89          // 30 pixeles más alta que la animación.
90          app.setSize( anim.getPreferredSize().width + 10,
91                       anim.getPreferredSize().height + 30 );
92          app.show();
93       } // fin de main
94    } // fin de la clase AnimadorLogo
```










**Figura 30.3** Animación de una serie de imágenes. (Parte 3 de 3.)

La clase **AnimadorLogo** carga un arreglo de **ImageIcons** en su constructor. Mientras se crea la instancia de cada objeto **ImageIcon** en la estructura **for** de la línea 21, el constructor **ImageIcon** carga una imagen para la animación (existen 30 imágenes en total) con la instrucción

```
imagenes[ i ] =
   new ImageIcon( "imagenes/deitel"  + i + ".gif" );
```

El argumento utiliza la concatenación de cadenas para ensamblar el nombre del archivo a partir de las partes **"imagenes/deitel"**, **i** y **".gif"**. Cada una de las imágenes de la animación se encuentra en uno de los archivos **"deitel0.gif"** a **"deitel29.gif"**. El valor de la variable de control de la estructura **for** se utiliza para seleccionar una de las 30 imágenes.

### Tip de rendimiento 30.1

*Es más eficiente cargar los marcos de la animación como una imagen, que cargar cada imagen por separado (puede utilizar un programa de dibujo para combinar los marcos de la animación dentro de la imagen). Si las imágenes se cargan desde la World Wide Web, cada imagen cargada requiere una conexión separada hacia el sitio en donde se almacenan las imágenes.*



### Tip de rendimiento 30.2

*Cargar todos los marcos de la animación como una imagen grande podría obligar a su programa a esperar para empezar a desplegar la animación.*

Después de cargar las imágenes, el constructor llama a **iniciaAnimacion** (definida en la línea 44) para comenzar la animación. La animación es controlada por una instancia de la clase ***Timer*** (del paquete **javax.swing**). Un objeto de la clase **Timer** genera **ActionEvent**s en un intervalo fijo en milisegundos (por lo general especificado como un argumento del constructor **Timer**) y notifica a todos sus **ActionListeners** registrados que ocurrió un evento. Las líneas 46 a 50

```
if ( cronoAnimacion == null ) {
   imagenActual = 0;
   cronoAnimacion = new Timer( retardoAnimacion, this );
   cronoAnimacion.start();
}
```

determinan si la referencia **cronoAnimacion** de **Timer** es **null**. Si es así, **imagenActual** se establece en 0 para indicar que la animación debe comenzar con la imagen del primer elemento del arreglo **imagenes**. La línea 48 asigna un nuevo objeto **Timer** a **cronoAnimacion**. El constructor **Timer** recibe dos argumentos, el retardo en milisegundos (en este ejemplo, el **retardoAnimacion** es 50 en milisegundos) y el **ActionListener** que responderá al **ActionEvent** de **Timer** (**this AnimadorLogo** implementa el **ActionListe-**

**ner** de la línea 8). La línea 49 inicia el objeto **Timer**. Una vez iniciado, **cronoAnimacion** generará un **ActionEvent** cada 50 milisegundos en este ejemplo. Las líneas 51 a 53

```
else  // continua desde la última imagen desplegada
   if ( ! cronoAnimacion.isRunning() )
      cronoAnimacion.restart();
```

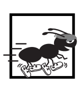
son para programas que pueden detener la animación y reiniciarla. Por ejemplo, para hacer de una animación "amigable para el navegador" en un applet, la animación debe detenerse cuando el usuario intercambia entre páginas Web. Si el usuario regresa a la página Web con la animación, es posible llamar al método **iniciaAnimacion** para reiniciar la animación. La condición **if** de la línea 52 utiliza el método **isRunning** de **Timer** para determinar si el **Timer** se está ejecutando actualmente (es decir, generando eventos). Si no se está ejecutando, la línea 53 llama al método **restart** de **Timer** para indicar que el **Timer** debe comenzar a generar eventos nuevamente.

En respuesta a cada uno de los eventos de **Timer** de este ejemplo, el método **actionPerformed** (línea 39) llama al método **repaint**. Esto programa una llamada al método **update** de **AnimadorLogo** (heredado desde la clase **JPanel**), el cual, a su vez, llama al método **paintComponent** de **AnimadorLogo** (línea 28). Recuerde que cualquier subclase de **JComponent** que realiza un dibujo debe hacerlo en su método **paintComponent**. Como mencionamos en el capítulo 29, la primera instrucción de cualquier método **paintComponent** debe ser una llamada al método **paintComponent** de la superclase, para garantizar que los componentes Swing se desplieguen correctamente. La condición **if** de las líneas 32 y 33

```
if ( imagenes[ imagenActual ].getImageLoadStatus() ==
   MediaTracker.COMPLETE ) {
```

utiliza el método ***getImageLoadStatus*** de **ImageIcon** para determinar si la imagen a desplegar está completamente cargada en memoria. Sólo las imágenes completas deben desplegarse, para hacer la animación tan suave como sea posible. Cuando la imagen está completamente cargada, el método regresa ***MediaTracker.COMPLETE***. Un objeto de la clase **MediaTracker** (del paquete **java.awt**) es utilizado por la clase **ImageIcon** para dar seguimiento a la carga de una imagen.

Cuando se cargan imágenes en un programa, dichas imágenes pueden registrarse con un objeto de la clase **MediaTracker**, para permitir al programa determinar cuándo una imagen se carga completamente. La clase **MediaTracker** también proporciona la habilidad de esperar la carga de una o varias imágenes, antes de permitir al programa continuar, y determina si ocurrió un error durante la carga de una imagen. Nosotros no necesitamos crear un **MediaTracker** de manera directa en este ejemplo, ya que la clase **ImageIcon** lo hace por nosotros. Sin embargo, cuando utilice la clase **Image** (como muestra la figura 30.1), es probable que quiera su propio **MediaTracker**.



### Tip de rendimiento 30.3

*Algunas personas que tienen experiencia con objetos **MediaTracker** han reportado que éstos tienen un efecto que va en detrimento del rendimiento. Mantenga esto en mente, como un área que analizará si necesita poner a punto sus aplicaciones multimedia.*



### Tip de rendimiento 30.4

*Utilizar el método **waitForAll** de **MediaTracker** para esperar a que todas las imágenes registradas se descarguen completamente puede resultar en un gran retraso una vez que el programa comienza la ejecución y hasta que las imágenes en realidad se despliegan. Entre más grandes sean las imágenes, mayor será el tiempo que el usuario tendrá que esperar. Utilice el método **waitForAll** sólo para esperar que un número pequeño de imágenes se desplieguen completamente.*

Si la imagen está completamente cargada, las líneas 34 y 35,

```
imagenes[ imagenActual ].paintIcon( this, g, 0, 0 );
imagenActual = ( imagenActual + 1 ) % totalImagenes;
```

dibujan el **ImageIcon** en el elemento **imagenActual** del arreglo y prepare la siguiente imagen a desplegar incrementando en 1 a **currentImage**. Observe el cálculo del módulo para garantizar que el valor de **currentImage** se establezca en 0, cuando se incremente a más de 29 (el último subíndice de elementos del arreglo).

El método **`stopAnimation`** (línea 56), detiene la animación con la línea 58,

```
cronoAnimacion.stop();
```

la cual utiliza el método ***`stop`*** de **`Timer`** para indicar que el **`Timer`** debe detener la generación de eventos. Esto, a su vez, previene que **`actionPerformed`** llame a **`repaint`** para iniciar el dibujo de la siguiente imagen del arreglo.

**Observación de ingeniería de software 30.1**

*Cuando genere una animación para utilizarla en un applet, proporcione un mecanismo para deshabilitarla cuando el usuario navegue una nueva página Web diferente a la página en la que el applet de la animación reside.*

Los métodos **`getMinimumSize`** (línea 61) y **`getPreferredSize`** (línea 66) se redefinen para ayudar al administrador de diseño a determinar el tamaño adecuado para un **`AnimadorLogo`** en un diseño. En este ejemplo, las imágenes son de 160 pixeles de ancho y de 80 pixeles de alto, por lo que el método **`getPreferredSize`** devuelve un objeto **`Dimension`** que contiene 160 y 80. El método **`getMinimumSize`** simplemente llama a **`getPreferredSize`** (una práctica común de programación). En **`main`** (línea 71), observe que el tamaño de la ventana de la aplicación se establece (líneas 90 y 91) en el mejor ancho de la animación más 10 pixeles, y en la mejor altura de la animación más 30 pixeles. Esto se debe a que el ancho y la altura de una ventana especifican los bordes externos de la ventana, no del *área del cliente* de la ventana (el área en donde pueden adjuntarse los componentes GUI).

En este ejemplo, pudimos aprovechar las diversas características que ayudan a producir animaciones suaves y controlables; los objetos **`ImageIcon`** cargaron las imágenes, un objeto de una subclase de **`JPanel`** desplegó las imágenes, y un objeto **`Timer`** controló la animación.

## 30.5 Tópicos de animación

Cuando ejecute la aplicación de la figura 30.3, podrá observar que la imagen se lleva tiempo en cargar. Si una animación no se diseña correctamente, esto con frecuencia da como resultado que las imágenes se desplieguen parcialmente. Es posible que usted vea que cada imagen se despliega por partes. Con frecuencia, esto es el resultado del formato que se utiliza para la imagen. Por ejemplo, las imágenes GIF pueden almacenarse en formatos *entrelazados* y *no entrelazados*. El formato indica el orden en el que se almacenan los pixeles de la imagen. Los pixeles de una imagen no entrelazada se almacenan en el mismo orden en el que los pixeles aparecen en la pantalla. Conforme se despliega una imagen no entrelazada, ésta aparece en pedazos de arriba hacia abajo, conforme se lee la información sobre los pixeles. Los pixeles de una imagen entrelazada se almacenan en filas de pixeles, sin embargo, las filas están en desorden. Por ejemplo, las filas de pixeles de la imagen pueden almacenarse en el orden 1, 5, 9, 13, ..., seguido por 2, 6, 10, 14, ..., y así sucesivamente. Cuando la imagen se despliega, ésta parece desvanecida, ya que el primer lote de filas presenta una imagen borrosa, y los lotes subsiguientes de filas mejoran la imagen desplegada, hasta que la totalidad de la imagen se completa. Para ayudar a evitar que aparezcan imágenes parciales en versiones anteriores de Java, dimos seguimiento a la carga de imágenes por medio de un objeto **`MediaTracker`**. Sólo se despliegan imágenes totalmente cargadas, para producir la animación más suave. Cada imagen a rastrear debe registrarse con el **`MediaTracker`**. Esto ahora se lleva a cabo por medio del constructor de la clase **`ImageIcon`**.



**Observación de ingeniería de software 30.2**

*La clase **`ImageIcon`** utiliza un objeto **`MediaTracker`** para determinar el estado de la imagen que está cargando.*



**Buena práctica de programación 30.2**

*En un applet, siempre despliegue algo mientras se cargan las imágenes. Entre más tiempo tenga que esperar un usuario para ver información en la pantalla, es más probable que abandone la página Web antes de que la información aparezca.*

Otro problema común con las animaciones es que la animación *parpadea* conforme cada imagen se despliega. Esto se debe a que se llama al método **`update`** en respuesta a cada **`repaint`**. En componentes GUI

de AWT, cuando **`update`** limpia el fondo del componente GUI, lo hace dibujando un rectángulo del tamaño del componente, relleno con el color de fondo actual. Esto cubre la imagen que se acababa de desplegar. Por lo tanto, la animación dibujaría una imagen, dormiría por una fracción de segundo, limpiaría el fondo (ocasionando un parpadeo), y dibujaría la siguiente imagen. En subclases **`JPanel`** de Swing (o de cualquier otro componente Swing), el método **`update`** se redefine para evitar limpiar el fondo, si el componente es transparente (el fondo se limpiará si el componente es opaco). Esto ayuda a eliminar el parpadeo.




### Observación de apariencia visual 30.1

*Los componentes Swing redefinen el método **`update`** para evitar que se limpie el fondo (en el caso de componentes transparentes), en respuesta a mensajes **`repaint`**.*

Si desea desarrollar aplicaciones basadas en multimedia, sus usuarios querrán audio y animaciones suaves. Las presentaciones disparejas son inaceptables. Esto con frecuencia ocurre cuando escribe aplicaciones que dibujan directamente en la pantalla. Otra técnica que se utiliza para producir animaciones suaves (y otros gráficos) es la de *gráficos con doble búfer*. Mientras el programa interpreta una imagen en la pantalla, éste puede construir la siguiente imagen en un *búfer fuera de pantalla*. Después, cuando es momento de que se despliegue la siguiente imagen, puede colocarla suavemente en la pantalla. Por supuesto, existe un *equilibrio espacio/tiempo*. La memoria adicional requerida puede ser importante, pero el rendimiento mejorado del despliegue lo vale.

Los gráficos con doble búfer también son útiles en programas que necesitan utilizar capacidades de dibujo en métodos diferentes de **`paint`** o **`paintComponent`** (en donde hemos hecho todos nuestros dibujos hasta este punto). El búfer fuera de pantalla puede pasarse entre métodos, o incluso entre objetos de diferentes clases, para permitir a otros métodos u objetos dibujar en el búfer fuera de pantalla. Los resultados del dibujo pueden entonces desplegarse en otro momento.



### Tip de rendimiento 30.5

*El doble búfer puede reducir o eliminar el parpadeo de una animación, pero puede disminuir visiblemente la velocidad a la que se ejecuta la animación.*

Cuando todos los pixeles de una imagen no se despliegan al mismo tiempo, una animación tiene más parpadeo. Cuando una imagen se dibuja por medio de gráficos con doble búfer, en el momento en que la imagen se despliega, ésta habrá sido dibujada fuera de la pantalla, y las imágenes parciales que el usuario normalmente vería, están ocultas para él. Todos los pixeles se desplegarán para el usuario en un "tris", para que el parpadeo se vea substancialmente disminuido, o para que desaparezca.

Los conceptos básicos de un gráfico con doble búfer son los siguientes: crear una **`Imagen`** en blanco, dibujar en la **`Imagen`** en blanco (utilizando métodos de la clase **`Graphics`**) y desplegar la imagen. La **`Imagen`** almacena los pixeles que se copiarán en la pantalla. La referencia **`Graphics`** se utiliza para dibujar los pixeles. Toda imagen tiene un contexto gráfico asociado; es decir, un objeto de la clase **`Graphics`** que permite que el dibujo se realice. Las referencias **`Image`** y **`Graphics`** utilizadas para los gráficos con doble búfer con frecuencia se conocen como *imagen fuera de la pantalla* y *contexto gráfico fuera de la pantalla*, debido a que en realidad no manipulan pixeles de pantalla.

Los componentes GUI de Swing se despliegan utilizando las capacidades de dibujo de Java. Por lo tanto, los componentes GUI de Swing están sujetos a muchos de los mismos problemas que se encuentran en una animación típica. De manera predeterminada, Swing utiliza gráficos con doble búfer para interpretar todos los componentes GUI. Al diseñar nuestro **`AnimadorLogo`** como una subclase de **`JPanel`**, podemos aprovechar los gráficos con doble búfer integrados de Swing para producir las animaciones más suaves.




### Observación de apariencia visual 30.2

*Los componentes GUI de Swing se interpretan utilizando gráficos con doble búfer, de manera predeterminada.*

## 30.6 Cómo personalizar applets por medio de la etiqueta `param` de HTML

Cuando navegue en la World Wide Web, con frecuencia encontrará applets que son del dominio público; puede utilizarlos de manera gratuita en sus propias páginas Web (normalmente como un intercambio por los créditos del creador del applet). Una característica común de dichos applets es la capacidad de personalizar el applet a

través de los parámetros que se proporcionan en el archivo HTML que invoca el applet. Por ejemplo, el siguiente código HTML del archivo AppletLogo.html

```
<html>
<applet code="AppletLogo.class" width=400 height=400>
<param name="imagenestotales" value="30">
<param name="nombreimagen" value="deitel">
<param name="retardoanimacion" value="200">
</applet>
</html>
```

invoca al applet **AppletLogo** (figura 30.4) y especifica tres parámetros. Las líneas de la *etiqueta* ***param*** deben aparecer entre las etiquetas **applet** inicial y final. Estos valores pueden entonces utilizarse para personalizar el applet. Cualquier número de etiquetas **param** puede aparecer entre las etiquetas **applet** inicial y final. Cada parámetro tiene un ***nombre*** y un ***valor***. El método **getParameter** de **Applet** se utiliza para obtener el **valor** asociado con un parámetro específico y devuelve el **valor** como una **String**. El argumento pasado a **getParameter** es una **String** que contiene el nombre del parámetro en la etiqueta **param**. Por ejemplo, la instrucción

```
parametro = getParameter( "retardoanimacion" );
```

obtiene el valor asociado con el parámetro **retardoanimacion**, y lo asigna a la referencia **parametro** de **String**. Si no hay una etiqueta **param** que contenga el parámetro especificado, **getParameter** devuelve **null**.

```java
// Figura 30.4: AnimadorLogo.java
// Animación de una serie de imágenes
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AnimadorLogo extends JPanel
                              implements ActionListener {
   protected ImageIcon imagenes[];
   protected int totalImagenes = 30,
                 imagenActual = 0,
                 retardoImagen = 50; // retardo de 50 milisegundos
   protected String nombreImagen = "deitel";
   protected Timer cronoAnimacion;

   public AnimadorLogo()
   {
      inicializaAnimacion();
   } // fin del constructor AnimadorLogo

   // constructor new para soportar la personalización
   public AnimadorLogo( int num, int retardo, String nombre )
   {
      totalImagenes = num;
      retardoImagen = retardo;
      nombreImagen = nombre;

      inicializaAnimacion();
   } // fin del constructor AnimadorLogo

```

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **param** de HTML; **AnimadorLogo.java**. (Parte 1 de 3.)

```
private void inicializaAnimacion()
{
   imagenes = new ImageIcon[ totalImagenes ];

   for ( int i = 0; i < imagenes.length; ++i )
      imagenes[ i ] = new ImageIcon( “imagenes/” +
                               nombreImagen + i + “.gif” );

   // se movió aquí para que getPreferredSize pueda verificar el tamaño de
   // la primera imagen cargada.
   setSize( getPreferredSize() );

   iniciaAnimacion();
} // fin del método inicializaAnimacion

public void paintComponent( Graphics g )
{
   super.paintComponent( g );

   if ( imagenes[ imagenActual ].getImageLoadStatus() ==
        MediaTracker.COMPLETE ) {
      imagenes[ imagenActual ].paintIcon( this, g, 0, 0 );
      imagenActual = ( imagenActual + 1 ) % totalImagenes;
   } // fin de if
} // fin del método paintComponent

public void actionPerformed( ActionEvent e )
{
   repaint();
} // fin del método actionPerformed

public void iniciaAnimacion()
{
   if ( cronoAnimacion == null ) {
      imagenActual = 0;
      cronoAnimacion = new Timer( retardoImagen, this );
      cronoAnimacion.start();
   }
   else  // continúa desde la última imagen desplegada
      if ( ! cronoAnimacion.isRunning() )
         cronoAnimacion.restart();
} // fin del método iniciaAnimacion

public void detieneAnimacion()
{
   cronoAnimacion.stop();
} // fin del método detieneAnimacion

public Dimension getMinimumSize()
{
   return getPreferredSize();
} // fin del método getMinimumSize

```

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **`param`** de HTML; **`AnimadorLogo.java`**. (Parte 2 de 3.)

```
   public Dimension getPreferredSize()
   {
      return new Dimension( imagenes[ 0 ].getIconWidth(),
                              imagenes[ 0 ].getIconHeight() );
   } // fin del método getPreferredSize

   public static void main( String args[] )
   {
      AnimadorLogo anim = new AnimadorLogo();

      JFrame app = new JFrame( "Prueba Animacion" );
      app.getContentPane().add( anim, BorderLayout.CENTER );

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            } // fin del método windowClosing
         } // y de la clase interna anónima
      ); // y de addWindowListener

      app.setSize( anim.getPreferredSize().width + 10,
                   anim.getPreferredSize().height + 30 );
      app.show();
   } // fin de main
} // fin de la clase AnimadorLogo
```

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **`param`** de HTML; **`AnimadorLogo.java`**. (Parte 3 de 3.)

```
// Figura 30.4: AppletLogo.java
// Personalización de un applet por medio de parámetros en HTML
//
// El parámetro HTML "retardoAnimacion" es un int que indica
// los milisegundos de retardo entre las imágenes (50 de manera
   predeterminada).
//
// El parámetro HTML "nombreimagen" es el nombre de la base de las imágenes
// que se desplegará (es decir, "deitel" es el nombre base de las
// imágenes "deitel0.gif," "deitel1.gif," etc.). El applet
// asume que las imágenes están en un subdirectorio "imagenes" del
// directorio en el cual reside el applet.
//
// El parámetro HTML "totalimagenes" es un entero que representa el
   número total
// de imágenes en la animación. El applet asume que las imágenes están
// numeradas desde 0 hasta totalimagenes - 1 (30 de manera predeterminada).

import java.awt.*;
import javax.swing.*;

public class AppletLogo extends JApplet{
```

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **`param`** de HTML; **`AppletLogo.java`**. (Parte 1 de 2.)

```
    public void init()
    {
       String parametro;

       parametro = getParameter( "retardoanimacion" );
       int retardoAnimacion = ( parametro == null ? 50 :
                               Integer.parseInt( parametro ) );

       String nombreImagen = getParameter( "nombreimagen" );

       parametro = getParameter( "totalimagenes" );
       int totalImagenes = ( parametro == null ? 0 :
                            Integer.parseInt( parametro ) );

       // Crea una instancia de AnimadorLogo
       AnimadorLogo animador;

       if ( nombreImagen == null || totalImagenes == 0 )
          animador = new AnimadorLogo();
       else
          animador = new AnimadorLogo( totalImagenes,
                        retardoAnimacion, nombreImagen );

       setSize( animador.getPreferredSize().width,
                animador.getPreferredSize().height );
       getContentPane().add( animador, BorderLayout.CENTER );

       animador.iniciaAnimacion();
    } // fin del método init
} // fin de la clase AppletLogo
```

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **`param`** de HTML; **`AppletLogo.java`**. (Parte 2 de 2.)

En la figura 30.4 modificamos la clase **`AnimadorLogo`** para poder utilizarla desde un applet y personalizarla a través de los parámetros del archivo HTML del applet. La clase **`AppletLogo`** permite a los diseñadores de páginas Web personalizar la animación para utilizarla en sus propias imágenes. Se proporcionan tres parámetros. El parámetro **`retardoAnimacion`** es el número de milisegundos a dormir entre las imágenes que se despliegan. Este valor se convertirá en un entero y se utilizará como el valor para la variable de instancia **`sleepTime`**. El parámetro **`nombreimagen`** es el nombre base de las imágenes a cargar. Esta **`String`** se asignará a la variable de instancia **`nombreImagen`**. El applet asume que las imágenes se encuentran en un subdirectorio llamado **`imagenes`** que puede localizarse en el mismo directorio del applet. El applet también asume que los nombres de los archivos de imágenes están numerados a partir de 0. El parámetro **`totalimagenes`** representa el número total de imágenes en la animación. Su valor se convertirá en un entero y se asignará a la variable de instancia **`totalImagenes`**.

La clase **`AnimadorLogo`** tiene diversas características nuevas para permitir su uso y su personalización en el **`AppletLogo`**. En la línea 13, se define la variable de instancia **`nombreImagen`**. Ésta almacenará el

nombre base predeterminado **"deitel"**, que es parte de todo nombre de archivo, o almacenará el nombre personalizado pasado al applet desde el documento HTML.

Ahora hay dos constructores; uno predeterminado (línea 16) y otro que toma argumentos para personalizar la animación (línea 22). Ambos constructores pueden llamar a nuestro nuevo método de utilidad **inicializaAnimacion** (línea 31) para cargar las imágenes e iniciar la animación. Las instrucciones de **inicializaAnimacion** estaban originalmente en el constructor predeterminado. La llamada al método **setSize** de la línea 41 (que se utiliza para preceder la carga de imágenes) se movió hacia la línea 41 para que el **AnimadorLogo** pudiera establecer un nuevo tamaño, de acuerdo con el ancho y el alto de la primera imagen de la animación. Para acomodar el nuevo tamaño basado en la primera imagen, el método **getPreferredSize** (línea 84) ahora devuelve un objeto **Dimension** que contiene el ancho y la altura de la primera imagen de la animación.

La clase **AppletLogo** (línea 130) define un método **init** en el que se leen los tres parámetros HTML con el método **getParameter** de **Applet** (líneas 135, 139 y 141). Después de que se leen los parámetros y de que los dos parámetros enteros se convierten en valores **int**, la estructura **if/else** de las líneas 148 a 152 crea un **AnimadorLogo**. Si el **nombreImagen** es **null**, o **totalImagenes** es **0**, se llama al constructor predeterminado **AnimadorLogo** y se utilizará la animación predeterminada. De lo contrario, **totalImagenes**, **retardoAnimación** y **nombreImagen** se pasan al constructor de tres argumentos **AnimadorLogo**, y éste utiliza dichos argumentos para personalizar la animación.

## 30.7 Mapas de imágenes

Una técnica común para crear páginas Web interesantes es el uso de *mapas de imágenes*. Un mapa de imágenes es una imagen que tiene *áreas sensibles* en donde el usuario puede hacer clic para realizar una tarea como cargar una página Web diferente en un navegador. Cuando el usuario posiciona el puntero del ratón sobre un área sensible, normalmente se despliega un mensaje descriptivo en el área de estado del navegador. Esta técnica puede utilizarse para implementar un sistema de *ayuda de burbuja*. Cuando el usuario posiciona el puntero del ratón sobre un elemento en particular de la pantalla, un sistema con ayuda de burbuja normalmente despliega un mensaje en una pequeña ventana que aparece sobre el elemento de la pantalla. En Java, el mensaje puede desplegarse en la barra de estado.

La figura 30.5 carga una imagen que contiene diversos iconos del *Java Multimedia Cyber Classroom*, el CD interactivo con la versión multimedia de este texto. Estos iconos pueden parecerle conocidos; están diseñados para imitar los iconos que utilizamos en este libro. El programa permite al usuario posicionar el puntero del ratón sobre un icono y desplegar un mensaje descriptivo para el icono. El manejador de eventos **mouseMoved** (línea 24) toma la coordenada *x* del ratón y la pasa al método **translateLocation** (línea 42). La coordenada *x* se evalúa para determinar el icono sobre el que se posicionó el ratón cuando se llamó al método **mouseMoved**. El método **translateLocation** entonces devuelve un mensaje que indica lo que el icono representa. Este mensaje se despliega en la barra de estado del **appletviewer** (o del navegador).

```
1  // Figura 30.5: MapaImagen.java
2  // Demostración de un mapa de imagenes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class MapaImagen extends JApplet {
8     private ImageIcon mapaImagen;
9     private int ancho, alto;
10
11    public void init()
12    {
13       addMouseListener(
14          new MouseAdapter() {
```

**Figura 30.5** Demostración de un mapa de imágenes. (Parte 1 de 3.)

```
        public void mouseExited( MouseEvent e )
        {
           showStatus( “Apuntador fuera del applet” );
        } // fin de método mouseExited
      } // fin de la clase interna anónima
); // fin de addMouseListener

    addMouseMotionListener(
      new MouseMotionAdapter() {
        public void mouseMoved( MouseEvent e )
        {
          showStatus( trasladaUbicacion( e.getX() ) );
        } // fin de método mouseMoved
      } // fin de la clase interna anónima
    ); // fin de addMouseMotionListener

    mapaImagen = new ImageIcon( “iconos2.gif” );
    ancho = mapaImagen.getIconWidth();
    alto = mapaImagen.getIconHeight();
    setSize( ancho, alto );
  } // fin del método init

  public void paint( Graphics g )
  {
    mapaImagen.paintIcon( this, g, 0, 0 );
  } // fin del método paint

  public String trasladaUbicacion( int x )
  {
    // determina el ancho de cada icono (existen 6)
    int anchoIcono = ancho / 6;

    if ( x >= 0 && x <= anchoIcono )
      return “Error comun de programacion”;
    else if ( x > anchoIcono && x <= anchoIcono * 2 )
      return “Buena practica de programacion”;
    else if ( x > anchoIcono * 2 && x <= anchoIcono * 3 )
      return “Tip de rendimiento”;
    else if ( x > anchoIcono * 3 && x <= anchoIcono * 4 )
      return “Tip de portabilidad”;
    else if ( x > anchoIcono * 4 && x <= anchoIcono * 5 )
      return “Observacion de ingenieria de software”;
    else if ( x > anchoIcono * 5 && x <= anchoIcono * 6 )
      return “Tip para prueba y depuracion”;

    return “”;
  } // fin del método trasladaUbicacion
} // fin de la clase MapaImagen
```







**Figura 30.5** Demostración de un mapa de imágenes. (Parte 2 de 3.)




**Figura 30.5** Demostración de un mapa de imágenes. (Parte 3 de 3.)

Hacer clic en este applet no ocasionará acción alguna. Si fuéramos a agregar capacidades de red, podríamos modificar este applet para permitir que cada icono estuviera asociado con una URL diferente.

## 30.8 Recursos en Internet y en la World Wide Web

Esta sección presenta diversos recursos en Internet y en la Web para sitios relacionados con multimedia.

**`http://www.nasa.gov/gallery/index.html`**
La *galería multimedia de la NASA* contiene una amplia variedad de imágenes, clips de audio y vídeo que puede descargar, para utilizarlos para probar sus programas multimedia en Java.

**`http://sunsite.sut.ac.jp/multimed/`**
La *Sunsite Japan Multimedia Collection* también proporciona una amplia variedad de imágenes, clips de audio y vídeo que puede descargar para fines educativos.

**`http://www.anbg.gov.au/anbg/index.html`**
El sitio Web *Australian National Botanics Gardens* proporciona vínculos hacia sonidos de muchos animales. Pruebe el vínculo *Common Birds*.

## *RESUMEN*

- El método **`getImage`** de **`Applet`** carga una **`Imagen`**. Una versión de **`getImage`** toma dos argumentos, una ubicación en donde se almacena el archivo y el nombre del archivo de la imagen.
- El método **`getDocumentBase`** de **`Applet`** devuelve la ubicación del archivo HTML del applet en Internet, como un objeto de la clase **`URL`** (del paquete **`java.net`**).
- Una **`URL`** almacena un Localizador Uniforme (o Universal) de Recursos; un formato estándar para una dirección de una pieza de información en Internet.
- Java soporta dos formatos de imagen, GIF (Formato de Intercambio de Gráficos) y JPEG (Grupo unido de expertos en fotografía). Los nombres de archivos para estos tipos terminan con **`.gif`** o **`.jpg`** (o **`.jpeg`**), respectivamente.
- La clase **`ImageIcon`** proporciona constructores que permiten a un objeto **`ImageIcon`** inicializarse con una imagen desde la computadora local, o con una imagen almacenada en un servidor Web en Internet.

- El método **Graphics** de **drawImage** recibe cuatro argumentos, una referencia al objeto **Image** en el cual se almacena la imagen, las coordenadas *x* y *y* en donde debe desplegarse la imagen y una referencia al objeto **ImageObserver**.
- Otra versión del método **drawImage** de **Graphics** despliega una imagen *a escala*. El cuarto y el quinto argumentos especifican el ancho y la altura de la imagen para propósitos del desplegado de dicha imagen.
- La interfaz **ImageObserver** se implementa mediante la clase **Component** (una superclase indirecta de **Applet**). A las **ImageObserver** se les notifica la actualización de una imagen que se despliega mientras se descarga el resto de la imagen.
- El método **paintIcon** de **ImageIcon** despliega la imagen de **ImageIcon**. El método requiere cuatro argumentos: una referencia al **Component** en el cual se desplegará la imagen, una referencia al objeto **Graphics** que se utiliza para interpretar la imagen, la coordenada *x* y *y* de la esquina superior izquierda de la imagen, y la coordenada *y* de la esquina superior izquierda de la imagen.
- El método **paintIcon** de la clase **ImageIcon** no permite escalar ninguna imagen. La clase proporciona el método **getImage** el cual devuelve una referencia a **Image** que puede utilizarse con el método **drawImage** de **Graphics** para desplegar una versión a escala de una imagen.
- El método **play** de **Applet** tiene dos formas:

```
public void play( URL ubicación, String nombreArchivoDeSonido );
public void play( URL URLdeSonido );
```

Una versión carga el clip de audio almacenado en el archivo **nombreArchivoDeSonido** desde la **ubicación** y reproduce el sonido. El otro toma una URL que contiene la ubicación y el nombre del archivo del clip de audio.

- El método **getDocumenBase** de **Applet** indica la ubicación del archivo HTML que cargó el applet. El método **getCodeBase** indica en dónde se localiza el archivo **.class** para el applet que se carga.
- El motor de audio que reproduce los clips de audio soporta varios formatos de audio que incluyen el formato de archivo de sonido de Sun (extensión **.au**), formato de archivo Wave de Windows (extensión **.wav**), el formato de archivo AIFF de Macintosh (extensión **.aif** o **.aiff**) y el formato de archivo Musical Instrument Digital Interface (MIDI) (extensión **.mid** o **.rmi**).
- El método **getAudioClip** de **Applet** tiene dos formas que toman los mismos argumentos que el método **play**. El método **getAudioClip** devuelve una referencia a un **AudioClip**. **AudioClip** tiene tres métodos, **play**, **loop** y **stop**. El método **play** reproduce una vez el sonido. El método **loop** repite de manera continua el clip de audio. El método **stop** termina un clip de audio que está en reproducción.
- Los objetos **Timer** generan **ActionEvents** en intervalos fijos en milisegundos y notifica a sus **ActionListener**s que ocurrieron los eventos. El constructor **Timer** recibe dos argumentos, el retardo en milisegundos y el **ActionListener**. El método **start** de **Timer** indica que **Timer** debe comenzar a generar eventos. El método **restart** de **Timer** indica que **Timer** debe comenzar nuevamente a generar eventos.
- El método **getImageLoadStatus** de **ImageIcon** determina si una imagen está cargada completamente en memoria. El método devuelve **MediaTracker.COMPLETE** si la imagen ya se cargó por completo.
- Las imágenes pueden registrarse con un objeto de la clase **MediaTracker** para permitir al programa determinar cuándo una imagen está cargada completamente.
- Las imágenes GIF pueden almacenarse en formatos entrelazados y no entrelazados. El formato indica el orden en el cual se almacenan los pixeles de la imagen. Mientras se despliega una imagen no entrelazada, los trozos de imagen aparecen de arriba hacia abajo mientras se lee la información de los pixeles. Los pixeles de una imagen entrelazada se almacenan en filas de pixeles, pero las filas están en desorden. Cuando se despliega la imagen, ésta parece desvanecida, ya que el primer lote de filas presenta una imagen borrosa, y los lotes subsiguientes de filas mejoran la imagen desplegada, hasta que la totalidad de la imagen se completa.
- Un problema común con las animaciones es que la animación parpadea al aparecer cada imagen. Por lo general, esto se debe a que se llama al método **update** en respuesta a cada **repaint**. En las subclases del **JPanel** de Swing (o cualquier otro componente de Swing), el método **update** se redefine para evitar la limpieza del fondo.
- Una técnica utilizada para producir animaciones suaves son los gráficos con doble búfer. Mientras el programa dibuja una imagen en la pantalla, puede construir la siguiente imagen en un búfer fuera de la pantalla. Entonces, cuando es tiempo de desplegar la siguiente imagen, ésta puede colocarse suavemente en la pantalla.
- Los componentes GUI de Swing se despliegan mediante el uso de las capacidades de dibujo de Swing. Por lo tanto, los componentes GUI de Swing están sujetos a muchos de los mismos problemas que se encuentran en una animación típica. De manera predeterminada, Swing utiliza doble búfer para interpretar todos los componentes GUI de Swing.

- Los applets pueden personalizarse mediante los parámetros (la etiqueta **<param>**) que se suministran en el archivo HTML que invoca al applet. Las líneas de la etiqueta **<param>** deben aparecer entre la etiqueta de **applet** de inicio y la etiqueta **applet** final. Cada parámetro tiene un **nombre** y un **valor**.
- El método **getParameter** de **Applet** obtiene el **valor** asociado con un parámetro específico y devuelve el **valor** como un **String**. El argumento se pasa a **getParameter** como un **String** que contiene el nombre del parámetro en la etiqueta **param**. Si no existe la etiqueta **param** que contiene el parámetro especificado, **getParameter** devuelve **null**.
- Un mapa de imágenes es una imagen que no tiene áreas sensibles en las cuales, el usuario puede hacer clic para llevar a cabo una tarea tal como la carga de una página Web diferente en un navegador.

## TERMINOLOGÍA

altura de una imagen
ancho de una imagen
animación
animación de una serie de imágenes
archivo AIFF de Macintosh (**.aif** o **.aiff**)
archivo de sonido de Sun (**.au**)
archivo HTML
archivo Wave de Windows (**.wav**)
área sensible de un mapa de imágenes
atributo de nombre de la etiqueta **param**
atributo **value** de la etiqueta
búfer fuera de pantalla
clase **Image**
clase **ImageIcon**
clase **MediaTracker**
clase **Timer**
clase **URL**
clip de audio
contexto gráfico fuera depantalla
equilibrio espacio/tiempo
escalar una imagen
etiqueta **param**
extensión de nombre de archivo **.aif**
extensión de nombre de archivo **.aiff**
extensión de nombre de archivo **.au**
extensión de nombre de archivo **.gif** gráficos
extensión de nombre de archivo **.jpeg**
extensión de nombre de archivo **.jpg**
extensión de nombre de archivo **.mid**
extensión de nombre de archivo **.rmi**
extensión de archivo **.wav**
formato de archivo de sonido de Sun (**.au**)
Formato de Intercambio de Gráficos (GIF)
gráficos con doble búfer
Grupo unido de expertos en fotografía (JPEG)
imágenes
imagen fuera de pantalla
imagen GIF entrelazada
imagen GIF no entrelazada
interfaz **ImageObserver**
Localizador Uniforme de Recursos (URL)
mapa de imágenes
método **drawImage** de la clase **Graphics**
método **getAudioClip** de la clase **Applet**
método **getCodeBase** de la clase **Applet**
método **getDocumentBase** de la clase **Applet**
método **getHeight** de la clase **Component**
método **getIconHeight** de la clase **ImageIcon**
método **getIconWidth** de la clase **ImageIcon**
método **getImage** de la clase **Applet**
método **getImage** de la clase **ImageIcon**
método **getImageLoadStatus**
método **getParameter** de la clase **Applet**
método **getWidth** de la clase **Component**
método **loop** de la interfaz **AudioClip**
**MediaTracker.COMPLETE**
método **paintIcon** de la clase **ImageIcon**
método **play** de la clase **Applet**
método **play** de la interfaz **AudioClip**
método **repaint** de la clase **Component**
método **restart** de la clase **Timer**
motor de audio
método **start** de la clase **Timer**
método **stop** de la clase **Timer**
método **stop** de la interfaz **AudioClip**
método **update** de la clase **Component**
multimedia
**param**
personalización de un applet
reducción del parpadeo de una animación
sistema de ayuda de burbuja
sonido

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

**30.1** Cuando reproduzca sonidos en un applet o en una aplicación, proporcione un mecanismo para que el usuario pueda deshabilitar el sonido.

**30.2** En un applet, siempre despliegue algo mientras se cargan las imágenes. Entre más tiempo tenga que esperar un usuario para ver información en la pantalla, es más probable que abandone la página Web antes de que la información aparezca.

## OBSERVACIONES DE APARIENCIA VISUAL

**30.1** Los componentes Swing redefinen el método **`update`** para evitar que se limpie el fondo (en el caso de componentes transparentes), en respuesta a mensajes **`repaint`**.

**30.2** Los componentes GUI de Swing se interpretan utilizando gráficos con doble búfer, de manera predeterminada.

## TIPS DE RENDIMIENTO

**30.1** Es más eficiente cargar los marcos de la animación como una imagen, que cargar cada imagen por separado (puede utilizar un programa de dibujo para combinar los marcos de la animación dentro de la imagen). Si las imágenes se cargan desde la World Wide Web, cada imagen cargada requiere una conexión separada hacia el sitio en donde se almacenan las imágenes.

**30.2** Cargar todos los marcos de la animación como una imagen grande podría obligar a su programa a esperar para empezar a desplegar la animación.

**30.3** Algunas personas que tienen experiencia con objetos **`MediaTracker`** han reportado que éstos tienen un efecto que va en detrimento del rendimiento. Mantenga esto en mente, como un área que analizará si necesita poner a punto sus aplicaciones multimedia.

**30.4** Utilizar el método **`waitForAll`** de **`MediaTracker`** para esperar a que todas las imágenes registradas se descarguen completamente puede resultar en un gran retraso una vez que el programa comienza la ejecución y hasta que las imágenes en realidad se despliegan. Entre más grandes sean las imágenes, mayor será el tiempo que el usuario tendrá que esperar. Utilice el método **`waitForAll`** sólo para esperar que un número pequeño de imágenes se desplieguen completamente.

**30.5** El doble búfer puede reducir o eliminar el parpadeo de una animación, pero puede disminuir visiblemente la velocidad a la que se ejecuta la animación.

## TIP DE PORTABILIDAD

**30.1** La clase **`Image`** es una clase **`abstract`**, por lo que no pueden crearse objetos de **`Image`** de manera directa. Para lograr la independencia de la plataforma, la implementación de Java en cada plataforma proporciona su propia subclase de **`Image`** para almacenar la información de la imagen.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

**30.1** Cuando genere una animación para utilizarla en un applet, proporcione un mecanismo para deshabilitarla cuando el usuario navegue una nueva página Web diferente a la página en la que el applet de la animación reside.

**30.2** La clase **`ImageIcon`** utiliza un objeto **`MediaTracker`** para determinar el estado de la imagen que está cargando.

## EJERCICIOS DE AUTOEVALUACIÓN

**30.1** Complete los espacios en blanco:

a) El método ____________ de **`Applet`** carga la imagen dentro de un applet.
b) El método ____________ de **`Applet`** devuelve como un objeto de la clase **`URL`** a la ubicación en Internet del archivo HTML que invocó al applet.
c) Una ____________ es un formato estándar para una dirección de una pieza de información en Internet.
d) El método ____________ de **`Graphics`** despliega una imagen de un objeto.
e) Con la técnica de ____________, mientras el programa interpreta una imagen en la pantalla, podría construir la siguiente imagen en un búfer fuera de pantalla. Entonces, cuando es tiempo para desplegar la siguiente imagen, ésta puede colocarse suavemente en la pantalla.
f) Conforme se despliega una imagen ____________, ésta aparece desvanecida mientras el primer lote de filas dibuja un borrador de la imagen y los lotes subsiguientes de filas refinan la imagen desplegada hasta que se completa la imagen.
g) Existen dos piezas clave para implementar un gráfico de doble búfer, una referencia a ____________ y una referencia a ____________. La primera es donde se desplegarán los pixeles reales; la segunda se utiliza para dibujar los pixeles.

h) Las imágenes pueden registrarse con un objeto ____________ para permitir al programa determinar cuando una imagen se cargó por completo.
i) Java proporciona dos mecanismos para reproducir sonidos en un applet, el método **`play`** de **`Applet`** y el método **`play`** de la *interfaz* ____________.
j) Un ____________ es una imagen que contiene *áreas sensibles* en las que el usuario puede hacer clic para llevar a cabo una tarea, tal como la carga de una página Web diferente.
k) El método ____________ de la clase **`ImageIcon`** despliega la imagen de **`ImageIcon`**.

**30.2** Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) En la actualidad, Java soporta dos formatos de imagen. Los nombres de archivos de estos tipos terminan con **`.jif`** o **`.gpg`** respectivamente.
b) Redefinir el método **`update`** del applet para llamar a **`paint`** sin limpiar el applet, reducirá significativamente el parpadeo de la animación.
c) Un sonido será depositado en la basura tan pronto como termine la reproducción.
d) Los componentes GUI de Swing contienen gráficos internos con doble búfer.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**30.1** a) **`getImage`**. b) **`getDocumentBase`**. c) **`URL`**. d) **`drawImage`**. e) Gráficos con doble búfer. f) Entrelazada. g) **`Image`**, **`Graphics`**. h) **`MediaTracker`**. i) **`AudioClip`**. j) Mapa de imágenes. k) **`paintIcon`**.

**30.2** a) Falso, debe ser **`.gif`** o **`.jpg`**. b) Verdadero. c) Falso, el sonido se marcará para el recolector de basura (si no está referenciado por un **`AudiClip`**) y se arrojará a la basura cuando el recolector de basura sea capaz de ejecutarse. d) Verdadero.

## EJERCICIOS

**30.3** Describa cómo hacer una animación “amigable para el navegador”.

**30.4** Explique los distintos aspectos de la eliminación del parpadeo en Java.

**30.5** Explique la técnica de los gráficos con doble búfer.

**30.6** Describa los métodos de Java para reproducir y manipular los clips de audio.

**30.7** (*Animación.*) Elabore un programa de animación en Java de propósito general. Su programa debe permitir al usuario especificar la secuencia de marcos a desplegar, la velocidad a la cual se despliegan las imágenes, los sonidos a reproducir mientras se ejecuta la aplicación, etcétera.

**30.8** (*Protector de pantalla.*) Utilice la animación de una serie de sus imágenes favoritas para crear un programa protector de pantalla. Elabore distintos efectos especiales que aprovechen la imagen, que la hagan girar, que la desvanezcan, que muevan la imagen hacia los límites de la pantalla y otras cosas similares.

**30.9** (*Borrar una imagen al azar.*) Suponga que se despliega una imagen en un área rectangular de la pantalla. Una manera de eliminar la imagen es establecer inmediatamente cada píxel con el mismo color, pero esto tiene un efecto visual monótono. Escriba un programa en Java que despliegue una imagen y que la elimine mediante la generación de números aleatorios para seleccionar los pixeles individuales a eliminar. Una vez que se eliminó la mayor parte de la imagen, elimine todos los pixeles restantes al mismo tiempo. Usted puede hacer referencia a los pixeles individuales haciendo que una línea comience y termine en el mismo punto. Puede intentar distintas variantes de este problema. Por ejemplo, podría desplegar las líneas de manera aleatoria, o podría desplegar las figuras al azar para eliminar regiones de la pantalla.

**30.10** (*Texto* i*ntermitente.*) Elabore un programa en Java que repita intermitentemente texto en la pantalla. Haga esto entremezclando un texto con una imagen plana de color como fondo. Permita al usuario controlar la “velocidad de parpadeo” y el color de fondo o patrón.

**30.11** (*Instantánea de imágenes.*) Elabore un programa en Java que coloque una instantánea de una imagen en la pantalla. Haga esto mediante la mezcla de una imagen con una imagen plana de color como fondo.

**30.12** (*Reloj digital.*) Implemente un programa que despliegue un reloj digital en la pantalla. Podría agregar opciones para escalar el reloj; desplegar el día, el mes y el año; emitir un sonido de alarma; reproducir ciertos sonidos en horas predefinidas y cosas similares.

**30.13** (*Llamar la atención hacia una imagen.*) Si usted desea enfatizar una imagen, puede colocar una fila simulada de bulbos de luz alrededor de la imagen. Puede dejar los bulbos encender y apagar al azar, o puede dejarlos encender y apagar uno después del otro.

**30.14** (*Zoom de imagen.*) Elabore un programa que le permita hacer acercamientos, o alejamientos de una imagen.



# Recursos en Internet y en Web

Este apéndice contiene una lista de valiosos recursos para C/C++ y Java en Internet y en World Wide Web. Estos recursos incluyen FAQs (preguntas más frecuentes), tutoriales, cómo obtener el C++ estándar de ANSI/ISO, información acerca de los compiladores más populares y cómo obtener compiladores gratuitos, demos, libros, tutoriales, herramientas de software, artículos, entrevistas, conferencias, diarios y revistas, cursos en línea, grupos de noticias y recursos profesionales.

Para mayor información acerca del American National Standards Institute (ANSI), o para adquirir los documentos de los estándares, visite a ANSI en **`www.ansi.org`**.

## A.1 Recursos para C/C++

**`sunir.org/booklist/`**
La Programmer´s Book List contiene una sección de libros de C++ con más de 30 títulos.

**`www.possibility.com/Cpp/CppCodingStandard.html`**
El sitio C++ Coding Standard contiene una extensa cantidad de información acerca de la programación en el lenguaje C++, así como una larga lista de recursos de C++ en la Web.

**`help-site.com/cpp.html`**
**`help-site.com`** proporciona vínculos a recursos de C++ en la Web.

**`www.glenmccl.com/tutor.htm`**
Este sitio es una buena referencia para los usuarios con conocimientos de C/C++. Los temas vienen acompañados con explicaciones detalladas y código de ejemplo.

**`www.programmersheaven.com/zone3/cat353/index.htm`**
Este sitio ofrece una extensa colección de bibliotecas para C++. Estas bibliotecas están disponibles para descargarlas de manera gratuita.

**`www.programmersheaven.com/zone3/cat155/index.htm`**
Éste es un sitio grandioso para los programadores, ya que ofrece muchas utilidades para C/C++.

**`www.programmersheaven.com/c/MsgBoard/wwwboard.asp?Board=3`**
Este sitio Web permite a los usuarios colocar preguntas y comentarios acerca de la programación en C/C++ para que otros usuarios los respondan.

**`www.hal9k.com/cug/`**
Este sitio proporciona recursos, diarios, software libre y otras cosas para C++.

**`www.codeguru.com/Cpp/Cpp/cpp_mfc/`**
Un popular sitio Web para programadores, **`codeguru.com`** proporciona una extensa lista de recursos para programadores que utilizan C y C++.

**www.dinkumware.com/refxc.html**
P.J. Plauger escribió el manual de referencia “Dinkum C Library”, y está disponible en la Web. Éste proporciona una referencia completa de todas las funciones y macros de la biblioteca estándar de C.

**www.devx.com/cplus/**
DevX es un recurso muy completo para programadores. Cada sección proporciona las últimas noticias, herramientas y técnicas para distintos lenguajes de programación. La sección C++ zone del sitio está dedicada a C++.

## A.2 Tutoriales de C++

**www.icce.rug.nl/documents/cplusplus/**
Este tutorial, escrito por un profesor universitario, está diseñado para los programadores en C que desean aprender a programar en C++.

**www.southeastmn.edu/Programs/computer/index.asp?drwID=14&dwinID=0**
El Minnesota State College Southeast Technical ofrece cursos en línea de C++ a crédito.

**www.cplusplus.com/doc/tutorial/**
Este tutorial cubre desde los fundamentos hasta la programación orientada a objetos avanzada con C++.

**www.cprogramming.com/tutorial.html**
Este sitio incluye un tutorial paso a paso que incluye código de ejemplo.

**www.programmersheaven.com/zone3/cat34/index.htm**
Este sitio contiene una lista de tutoriales organizados por temas. El rango de niveles de los tutoriales va desde principiante hasta experto.

## A.3 Preguntas frecuentes de C/C++

**www.cs.ruu.nl/wais/html/na-dir/C-faq/diff.html**
Este sitio Web contiene actualizaciones y modificaciones al FAQ **com.lang.c** (**www.eskimo.com/~scs/C-faq/top.html**).

**www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.html**
Este sitio consiste en una serie de vínculos a FAQs y tutoriales reunidos en el grupo de noticias de **comp.lang.c++**.

## A.4 comp.lang.c++

**www.research.att.com/~bs/homepage.html**
Ésta es la página personal de Bjarne Stroustrup, diseñador del lenguaje de programación C++. Él proporciona una lista de los recursos de C++, FAQs y otra información útil de acerca de C++.

**www.austinlinks.com/CPlusPlus/**
Este sitio cuenta con una lista de recursos para C++, la cual incluye sugerencias de libros, recursos profesionales, información acerca del lenguaje de programación C++ y vínculos a sitios con listas de recursos para C++.

**www.cyberdiem.com/vin/learn.html**
Learn C/C++ Today es el título de este sitio, el cual proporciona un número de tutoriales de gran alcance para C/C++.

**www.experts-exchange.com/Programming_Languages/cplusplus/**
El Experts Exchange es un recurso gratuito para profesionales en alta tecnología que desean compartir información con sus colegas. Los miembros pueden colocar sus preguntas y respuestas en este sitio.

**cplus.about.com/compute/cplus/**
Éste es el sitio **About.com** de los lenguajes de programación C/C++. Usted encontrará tutoriales, software libre, diccionarios, empleos, revistas y muchos otros elementos relacionados.

**news:comp.lang.c++**
Éste es un grupo de noticias dedicado a temas sobre el lenguaje de programación orientado a objetos C++.

**news:comp.lang.c++.moderated**
Éste es un grupo de noticias más dedicado técnicamente al lenguaje C++.

## A.5 Compiladores de C/C++

**ftp://gcc.gnu.org/pub/gcc/releases/index.html**
Un índice muy completo de las versiones gratuitas más recientes de GCC (en C++, y también en Java).

**www.comeaucomputing.com/features.html**
Comeau Computing ofrece su compilador gratuito, el cual soporta algunas características de C99.

**www.compilers.net/**
**Compilers.net** es un sitio diseñado para ayudarle a encontrar compiladores.

**msdn.microsoft.com/visualc/**
La página de Visual C++ de Microsoft proporciona información acerca del producto, resúmenes, información adicional e información para ordenar el compilador Visual C++.

**www.metrowerks.com/MW/Develop/Desktop/Windows/default.htm**
Metrowerks Code Warrior es un entorno de desarrollo para escribir código en C/C++ o Java.

**www.faqs.org/faqs/by-newgroup/comp/comp.compilers.html**
Ésta es una suite que contiene una lista de FAQs generadas dentro del grupo de noticias **comp.compilers**.

**www.borland.con/cbuilder/**
Éste es un vínculo hacia Borland C++ Builder 6. Una versión gratuita del compilador en línea de comando, disponible para su descarga.

**sunset.backbone.olemiss.edu/%7Ebobcook/eC/**
Este compilador de C++ está diseñado para usuarios que inician con C++ y que desean hacer la transición de Pascal a C++.

**www.intel.com/software/products/compilers/cwin/**
El compilador de C++ de Intel. Las plataformas que soporta incluyen Windows98, NT, 2000 y XP.

## A.6 Recursos para Java

**java.sun.com**
El sitio Web de Sun Microsystems es una parada esencial cuando buscamos información acerca de Java en la Web. Vaya a este sitio para descargar el Java2 Software Development Kit (J2SDK). Además, este sitio es un recurso completo que cuenta con noticias, información, soporte en línea, ejemplos de código y mucho más.

**http://www.developer.com/java/**
Gamelan, quien ahora es parte de **developer.com**, ha sido un grandioso recurso para Java desde sus inicios. El sitio de Gamelan se llama a sí mismo "El directorio oficial de Java". Este sitio originalmente era un gran repositorio de Java, en donde los individuos intercambiaban ideas sobre Java y ejemplos de programación en Java. Una de sus primeras ventajas era el volumen de código fuente disponible para mucha gente que estaba aprendiendo Java. En la actualidad es un recurso completo con referencias de Java, descargas gratuitas, áreas en donde puede hacer preguntas a los expertos en Java, grupos de discusión sobre Java, un glosario de la terminología relacionada con Java, eventos próximos relacionados con Java, directorios especializados en temas de la industria y cientos de recursos para Java.

**www.jars.com**
Otro sitio Web de **developer.com** es JARS; originalmente llamado el Java Applet Rating Service. El sitio JARS se denomina a sí mismo el "Servicio de Información #1 de Java". Originalmente, el sitio era un gran repositorio para los applets de Java. Su principal beneficio era que clasificaba cada applet registrado en el sitio como top 1%, top 5%, y top 25%, de manera que usted podía ver de inmediato los mejores applets de la Web. Cuando comenzaba el desarrollo del lenguaje Java, tener su applet en la clasificación anterior era una importan-

te forma de demostrar sus habilidades de programación en Java. Actualmente, JARS es otro sitio completo de recursos en Java. Muchos de los recursos de este sitio, así como los de **Gamelan** y **developer.com** están compartidos ya que estos sitios pertenecen a **EarthWeb**.

**http://www.java.sun.com/developer**
Éste es uno de los sitios Web de Sun Microsystems para Java. Este sitio gratuito tiene cerca de un millón de miembros. El sitio incluye soporte técnico, foros de discusión, cursos de entrenamiento en línea, artículos técnicos, anuncios acerca de las nuevas características de Java, acceso a nuevas tecnologías de Java, y vínculos hacia otros sitios importantes Web de Java. Aún cuando el sitio Web es gratuito, debe registrarse para poder utilizarlo.

**javawoman.com/index.html**
El sitio Web Java Woman tiene una de las listas más extensas de vínculos relacionados con Java que hemos encontrado en la Web. Usted encontrará listas de vínculos hacia libros de Java, entornos integrados de desarrollo, FAQs, ejemplos, documentación, tutoriales, herramientas y temas avanzados.

**www.nikos.com/javatoys/**
El sitio Web Java Toys incluye vínculos hacia las últimas noticias acerca de Java, Grupos de Usuarios de Java (GUJs), FAQs, herramientas, listas de correo relacionadas con Java, libros y documentación.

**www.devx.com/java/**
El sitio Development Exchange Java Zone incluye grupos de discusión acerca de Java, las noticias recientes acerca de Java, así como muchos otros recursos acerca de Java.

**www.acme.com/java/**
Esta página es un applet animado de Java del que se proporciona el código fuente. Este sitio es un excelente recurso de información sobre Java. La página proporciona software, notas y una lista de todos los vínculos hacia otros recursos. Bajo "software", usted encontrará algunos applets animados, clases de utilidad y aplicaciones.

**http://www-106.ibm.com/developerworks/subscription/downloads/**
El sitio IBM Developers Java Technology Zone lista las noticias más recientes, herramientas, ejemplos prácticos y eventos relacionados con IBM y Java.

## A.7 Productos de Java

**java.sun.com/products/**
Descargue el Java 2 SDK y otros productos relacionados con Java.

**wwws.sun.com/software/sundev/jde/index.html**
El IDE Sun One Studio es un ambiente de programación visual, que puede personalizarse de manera independiente de la plataforma.

**www.borland.com/jbuilder/**
La página de inicio del JBuilder de Borland contiene noticias, información del producto y soporte al cliente.

**http://www-306.ibm.com/software/awdtools/studiositedev/**
Descargue o lea más acerca de IBM WebSphere Studio para el ambiente de desarrollo en Java.

**www.metrowerks.com/MW/Develop/Desktop/Windows/default.htm**
El IDE CodeWarrior de Metrowerks soporta algunos lenguajes de programación, incluso Java.

## A.8 FAQs de Java

**javawoman.com/index.html**
El sitio Web Java Woman tiene una de las listas de vínculos relacionados con Java más extensas que encontramos en la Web. Usted encontrará listas de vínculos hacia libros de Java, entornos integrados de desarrollo, FAQs, ejemplos, documentación, tutoriales, herramientas y temas avanzados.

**www.nikos.com/javatoys/**
El sitio Web Java Toys incluye vínculos hacia las últimas noticias acerca de Java, Grupos de Usuarios de Java (GUJs), FAQs, ejemplos, documentación, tutoriales, herramientas y temas avanzados.

**www.devx.com./java**

El sitio Development Exchange Java Zone incluye foros de discusión relacionados con Java, noticias recientes de Java, así como muchos otros recursos de Java.

**www.ibiblio.org/javafaq/**

Este sitio proporciona las últimas noticias acerca de Java. Además contiene valiosos recursos de Java, que incluyen la lista de Java, un tutorial llamado Brewing Java, grupos de usuarios de Java, vínculos de Java, la lista de libros de Java, los Java Trade Shows, entrenamiento y ejercicios.

## A.9 Tutoriales de Java

**java.sun.com/docs/books/tutorial/**

El sitio Java Tutorial contiene varios tutoriales que incluyen una sección sobre JavaBeans, JDBC, RMI, Servlets, colecciones y la Java Native Interface.

**javawoman.com/index.html**

El sitio Web Java Woman contiene una de las listas de vínculos relacionadas con Java más extensas que hemos encontrado en la Web. Usted encontrará listas de vínculos hacia libros de Java, entornos integrados de desarrollo, FAQs, ejemplos, documentación, herramientas y temas avanzados.

**www.ibiblio.org/javafaq/**

Este sitio proporciona las noticias más recientes de Java. Además contiene recursos útiles de Java, los cuales incluyen la lista de preguntas más frecuentes (FAQs) de Java, un tutorial llamado Brewing Java, grupos de usuarios, ligas relacionadas con Java, la lista de libros de Java, Java Trade Shows, cursos de entrenamiento en Java y ejercicios.

## A.10 Revistas de Java

**www.javaworld.com**

JavaWorld, una revista en línea, es un excelente recurso para obtener información actualizada con respecto a Java. Usted encontrará nuevos tips, información acerca de conferencias y vínculos hacia sitios relacionados con Java.

**www.sys-con.com/java/**

Entérese de las últimas noticias acerca de Java en el sitio *Java Developer's Journal*. Esta revista es uno de los principales recursos para obtener noticias de Java.

**www.javareport.com**

El *Java Report* es un gran recurso para los desarrolladores en Java. Usted encontrará las últimas noticias relacionadas con la industria, códigos de ejemplo, listas de eventos, productos y empleos.

## A.11 Applets de Java

**java.sun.com**

Existe un gran número de applets de Java disponibles en la Web. El mejor lugar para comenzar es en la fuente: el sitio Web de Java de Sun Microsystems Inc. En la esquina superior izquierda de la página Web podemos encontrar un vínculo hacia la página Web de Applets de Sun.

**java.sun.com/applets/index.html**

Esta página contiene gran variedad de recursos para applets, incluso applets gratuitos que puede utilizar en su propio sitio Web, los applets de demostración del J2SDK, y una gran variedad de applets adicionales (muchos de los cuales pueden descargarse y utilizarse en su propia computadora). También existe una sección titulada "Applets at Work" en donde puede leer acerca de los usos de los applets en la industria.

**gamelan.com (www.developer.com/java/)**

Gamelan, quien ahora es parte de **developer.com**, ha sido un grandioso recurso para Java desde sus inicios. El sitio de Gamelan se llama a sí mismo "El directorio oficial de Java". Este sitio originalmente era un gran repositorio de Java, en donde los individuos intercambiaban ideas y ejemplos de programación en Java. Una de sus primeras ventajas era el volumen de código fuente disponible para mucha gente que estaba aprendiendo Java. En la actualidad es un recurso muy completo con referencias de Java, descargas gratuitas de Java, áreas en donde puede hacer preguntas a los expertos en Java, grupos de discusión sobre Java, un glosario de la terminología relacionada con Java, eventos próximos relacionados con Java, directorios especializados en temas de la industria y cientos de recursos para Java.

**www.jars.com**

Otro sitio Web de **developer.com** es JARS; originalmente llamado el Java Applet Rating Service. El sitio JARS se denomina a si mismo el "Servicio de Información #1 de Java". Originalmente el sitio era un gran repositorio para los applets de Java. Su principal beneficio era que clasificaba cada applet registrado en el sitio como top 1%, top 5%, y top 25%, de manera que usted podía ver de inmediato los mejores applets de la Web. Cuando comenzaba el desarrollo del lenguaje Java, tener su applet en la clasificación anterior era una importante forma de demostrar sus habilidades de programación en Java. En la actualidad, JARS es otro sitio completo de recursos en Java. Muchos de los recursos de este sitio, así como los de **Gamelan** y **developer.com**, están compartidos ya que estos sitios pertenecen a **EarthWeb**.

## A.12 Multimedia

**java.sun.com/products/java-media/jmf/**

Es la página de inicio del Java Media Framework en el sitio Web de Java. Aquí puede descargar la implementación más reciente del JMF. Además, el sitio contiene documentación para la JMF.

**www.nasa.gov/multimedia/highlights/index.html**

La galería multimedia de la NASA contiene una amplia variedad de imágenes, clips de audio y de video que puede descargar y utilizar para probar sus programas multimedia en Java.

**sunsite.sut.ac.jp/multimed/**

La Sunsite Japan Multimedia Collection también proporciona una amplia variedad de clips de audio y de video que puede descargar con propósitos educativos.

## A.13 Grupos de noticias de Java

**news:comp.lang.java**

**news:comp.lang.java.advocacy**

**news:comp.lang.java.announce**

**news:comp.lang.java.beans**

**news:comp.lang.java.corba**

**news:comp.lang.java.databases**

**news:comp.lang.java.gui**

**news:comp.lang.java.help**

**news:comp.lang.java.machine**

**news:comp.lang.java.programmer**

**news:comp.lang.java.softwaretools**

**news:cz.comp.lang.java**

**news:fj.comp.lang.java**



# Recursos en Internet y en Web para C99

Este apéndice contiene una lista de recursos para C99 en Internet y en la World Wide Web. Estos recursos incluyen FAQs (preguntas más frecuentes), tutoriales, cómo obtener el C99 estándar de ANSI/ISO, demos, libros, herramientas de software, artículos, entrevistas, conferencias, diarios y revistas, cursos en línea, grupos de noticias y recursos profesionales.

C99 es el estándar de ANSI más reciente del lenguaje de programación C. Fue desarrollado para que el lenguaje C evolucionara y así mantuviera el ritmo con el poderoso hardware actual y con los cada vez más demandantes requerimientos del usuario. El estándar de C99 es más capaz (que las primeras versiones de C) de competir con lenguajes como FORTRAN para aplicaciones matemáticas. Las capacidades de C99 incluyen el tipo **`long long`** para máquinas de 64 bits, números complejos para aplicaciones de ingeniería y un gran soporte para la aritmética de punto flotante. Además, C99 hace más consistente a C respecto a C++ al permitir el polimorfismo a través de funciones matemáticas con tipos genéricos, y a través de la creación de un tipo booleano definido.

El estándar de C99 contiene muchas modificaciones respecto a las primeras versiones del lenguaje. Éstas incluyen la funcionalidad avanzada para tipos de variables de punto flotante, booleanas y **`long long`**, la eliminación del **`int`** implícito y la posibilidad de definir variables en el encabezado de un ciclo **`for`**. Las explicaciones detalladas de todas las modificaciones a C99 las puede encontrar en el documento del estándar de ANSI/ISO y en muchos de los vínculos que aparecen más adelante.

Todavía no son muchos los compiladores disponibles que cumplen con C99. Algunas bibliotecas correspondientes a compiladores de C que soportan el nuevo estándar incluyen la biblioteca de C99 Dinkumware (**`www.dinkumware.com`**) y el compilador de C99 de Comeau Computing (**`www.comeaucomputing.com`**).

Puede adquirir el documento del estándar internacional para C99 en el American National Standards Institute (**`www.ansi.org`**). Puede descargar una lista de fe erratas del estándar. El International Committee for Information Technology Standards (INCITS) funge como el grupo de consultores técnicos de ANSI para el ISO/IEC Joint Technical Committee 1. Puede adquirir la documentación de C99 desde su sitio Web, **`www.incits.org`**.

## B.1 Recursos para C99

**`www.ansi.org`**
Todos los documentos de ANSI, incluso el estándar de C99, pueden encontrarse y adquirirse en este sitio.

**`www.incits.org/tc_home/j11.htm`**
Este sitio Web documenta el progreso del INCITS (InterNacional Committee for Information Technology Standards) en el desarrollo del estándar de C.

**anubis.dkuug.dk/JTC1/SC22/WG14/**
ISO/IEC JTC1/SC22/WG14 es el grupo de trabajo para la estandarización internacional del lenguaje de programación C. Aquí puede encontrar las últimas actualizaciones y revisiones.

**wwwold.dkuug.dk/JTC1/SC22/WG14/www/newinc9x.htm**
Contiene una lista de las características de C99.

**www.comeaucomputing.com/features.html**
Comeau Computing ofrece su compilador gratuito, el cual soporta muchas características de C99.

**www.dinkumware.com/libraries_ref.html**
Dinkumware ofrece licencias para las bibliotecas de C y C++ que cumplen con los estándares de ANSI y proporciona documentación en línea.

**www.thefreecountry.com/compilers/cpp.shtml**
Este sitio Web lista muchos compiladores gratuitos de C y C++ que incluyen algunos que cumplen con C99.

**david.tribble.com/text/cdiffs.htm**
David R. Tribble explica la compatibilidad entre C99 y el C++ de ANSI/ISO.

**gcc.gnu.org/c9xstatus.html**
Este sitio Web lista las características más recientes de C99 soportadas por la GNU Compiler Collection (GCC).

**www.cs.ruu.nl/wais/html/na-dir/C-faq/diff.html**
Este sitio Web contiene actualizaciones y modificaciones al sitio de FAQs **comp.lang.c**, el cual puede encontrar en **www.eskimo.com/~scs/C-faq/top.html**.

**www-ccs.ucsd.edu/c/**
Este sitio Web es una referencia integral para programar en C estándar. Contiene y documenta todas las bibliotecas estándar.

**www.lysator.liu.se/c/q8/index.html**
Doug Gwyn proporciona un ejemplo de las bibliotecas de C99 para el dominio público.

**www.ramtex.dk/standard/iostand.htm**
Una propuesta para la revisión de asuntos relacionados con el direccionamiento del hardware de entrada/salida en C99.

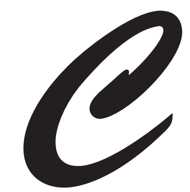
**home.att.net/~jackklein/c/standards.html**
Respuestas a FAQs acerca de ANSI e ISO, y por qué son importantes los estándares de C y C++.

**www.cl.cam.ac.uk/~mgk25/c-time/**
Una nueva biblioteca propuesta, **time**, para el nuevo anteproyecto de C.

**www.devworld.apple.com/tools/mpw-tools/c9x.html**
Este sitio Web contiene el documento oficial del Comité de C99.

**www.eskimo.com/~scs/C-faq/top.html**
Esta lista de FAQs contiene temas tales como apuntadores, asignación de memoria y cadenas.

**gcc.gnu.org/ml/gcc/**
Grupo de noticias sobre GNU que cubre muchos temas, tales como el C99 estándar.



# Tablas de precedencia de operadores

Los operadores aparecen en orden decreciente de precedencia, de arriba hacia abajo.

| Operador de C | Tipo | Asociatividad |
|---|---|---|
| () | paréntesis (operador de llamada a función) | izquierda a derecha |
| [] | subíndice de arreglo | |
| . | selección de miembros mediante un objeto | |
| -> | selección de miembros mediante un apuntador | |
| ++ | operador unario de preincremento | derecha a izquierda |
| -- | operador unario de predecremento | |
| + | suma unaria | |
| - | resta unaria | |
| ! | negación lógica unaria | |
| ~ | complemento unario a nivel de bits | |
| ( *tipo* ) | conversión de tipo al estilo C | |
| * | desreferencia | |
| & | dirección | |
| sizeof | determina un tamaño en bytes | |
| * | multiplicación | izquierda a derecha |
| / | división | |
| % | módulo | |
| + | suma | izquierda a derecha |
| - | resta | |
| << | desplazamiento a la izquierda a nivel de bits | izquierda a derecha |
| >> | desplazamiento a la derecha a nivel de bits | |
| < | menor que relacional | |
| <= | menor o igual que relacional | |

**Figura C.1** Tabla de precedencia de operadores en C. (Parte 1 de 2.)

| Operador de C | Tipo | Asociatividad |
|---|---|---|
| > | mayor que relacional | izquierda a derecha |
| >= | mayor o igual que relacional | |
| == | igual que relacional | izquierda a derecha |
| != | no es igual que | |
| & | AND a nivel de bits | izquierda a derecha |
| ^ | OR excluyente a nivel de bits | izquierda a derecha |
| \| | OR incluyente a nivel de bits | izquierda a derecha |
| && | AND lógico | izquierda a derecha |
| \|\| | OR lógico | izquierda a derecha |
| ?: | condicional ternario | derecha a izquierda |
| = | asignación | derecha a izquierda |
| += | asignación de suma | |
| -= | asignación de resta | |
| *= | asignación de multiplicación | |
| /= | asignación de división | |
| %= | asignación de módulo | |
| &= | asignación de AND a nivel de bits | |
| ^= | asignación de OR excluyente a nivel de bits | |
| \|= | asignación de OR incluyente a nivel de bits | |
| <<= | asignación de desplazamiento a la izquierda a nivel de bits | |
| >>= | asignación de desplazamiento a la derecha a nivel de bits | |
| , | coma | izquierda a derecha |

**Figura C.1** Tabla de precedencia de operadores en C. (Parte 2 de 2.)

| Operador de C++ | Tipo | Asociatividad |
|---|---|---|
| `::` | operador binario de resolución de alcance | izquierda a derecha |
| `::` | operador unario de resolución de alcance | |
| `()` | paréntesis (operador de llamada a función) | izquierda a derecha |
| `[]` | subíndice de arreglo | |
| `.` | selección de miembros mediante un objeto | |
| `->` | selección de miembros mediante un apuntador | |
| `++` | operador unario de postincremento | |
| `--` | operador unario de postdecremento | |
| `typeid` | información de tipo en tiempo de ejecución | |
| **dynamic_cast< tipo >** | conversión de tipo verificada en tiempo de ejecución | |
| **static_cast< tipo >** | conversión de tipo verificada en tiempo de compilación | |
| **reinterpret_cast< tipo >** | conversiones de tipo no estándares | |
| **const_cast< tipo >** | conversión de tipo para eliminar la constancia | |
| `++` | operador unario de preincremento | derecha a izquierda |
| `--` | operador unario de predecremento | |
| `+` | suma unaria | |
| `-` | resta unaria | |
| `!` | negación lógica unaria | |
| `~` | complemento unario a nivel de bits | |
| **(** *tipo* **)** | conversión de tipo al estilo C | |
| `sizeof` | determina un tamaño en bytes | |
| `&` | dirección | |
| `*` | desreferencia | |
| `new` | asignación dinámica de memoria | |
| `new[]` | asignación dinámica de arreglos | |
| `delete` | liberación automática de memoria | |
| `delete[]` | liberación automática de arreglos | |
| `.*` | apuntador a un miembro mediante un objeto | izquierda a derecha |
| `->*` | apuntador a un miembro mediante un apuntador | |
| `*` | multiplicación | izquierda a derecha |
| `/` | división | |
| `%` | módulo | |
| `+` | suma | izquierda a derecha |
| `-` | resta | |
| `<<` | desplazamiento a la izquierda a nivel de bits | izquierda a derecha |
| `>>` | desplazamiento a la derecha a nivel de bits | |
| `<` | menor que relacional | izquierda a derecha |
| `<=` | menor o igual que relacional | |
| `>` | mayor que relacional | |
| `>=` | mayor o igual que relacional | |

**Figura C.2** Tabla de precedencia de operadores en C++. (Parte 1 de 2.)

<table>
<tr><th>Operador de C++</th><th>Tipo</th><th>Asociatividad</th></tr>
<tr><td>==</td><td>igual que relacional</td><td>izquierda a derecha</td></tr>
<tr><td>!=</td><td>no es igual que relacional</td><td></td></tr>
<tr><td>&</td><td>AND a nivel de bits</td><td></td></tr>
<tr><td>^</td><td>OR excluyente a nivel de bits</td><td>izquierda a derecha</td></tr>
<tr><td>|</td><td>OR incluyente a nivel de bits</td><td>izquierda a derecha</td></tr>
<tr><td>&&</td><td>AND lógico</td><td>izquierda a derecha</td></tr>
<tr><td>||</td><td>OR lógico</td><td>izquierda a derecha</td></tr>
<tr><td>?:</td><td>condicional ternario</td><td>derecha a izquierda</td></tr>
<tr><td>=</td><td>asignación</td><td>derecha a izquierda</td></tr>
<tr><td>+=</td><td>asignación de suma</td><td></td></tr>
<tr><td>-=</td><td>asignación de resta</td><td></td></tr>
<tr><td>*=</td><td>asignación de multiplicación</td><td></td></tr>
<tr><td>/=</td><td>asignación de división</td><td></td></tr>
<tr><td>%=</td><td>asignación de módulo</td><td></td></tr>
<tr><td>&=</td><td>asignación de AND a nivel de bits</td><td></td></tr>
<tr><td>^=</td><td>asignación de OR excluyente a nivel de bits</td><td></td></tr>
<tr><td>|=</td><td>asignación de OR incluyente a nivel de bits</td><td></td></tr>
<tr><td><<=</td><td>asignación de desplazamiento a la izquierda a nivel de bits</td><td></td></tr>
<tr><td>>>=</td><td>asignación de desplazamiento a la derecha a nivel de bits</td><td></td></tr>
<tr><td>,</td><td>coma</td><td>izquierda a derecha</td></tr>
</table>

**Figura C.2** Tabla de precedencia de operadores en C++. (Parte 2 de 2.)

| Operador de Java | Tipo | Asociatividad |
|---|---|---|
| `++` | operador unario de postincremento | derecha a izquierda |
| `--` | operador unario de postdecremento | |
| `++` | operador unario de preincremento | derecha a izquierda |
| `--` | operador unario de predecremento | |
| `+` | suma unaria | |
| `-` | resta unaria | |
| `!` | negación lógica unaria | |
| `~` | complemento unario a nivel de bits | |
| `(` *tipo* `)` | conversión de tipo | |
| `*` | multiplicación | izquierda a derecha |
| `/` | división | |
| `%` | módulo | |
| `+` | suma | izquierda a derecha |
| `-` | resta | |
| `<<` | desplazamiento a la izquierda a nivel de bits | izquierda a derecha |
| `>>` | desplazamiento a la derecha a nivel de bits con extensión de signo | |
| `>>>` | desplazamiento a la derecha a nivel de bits con extensión de cero | |
| `<` | menor que relacional | |
| `<=` | menor o igual que relacional | |
| `>` | mayor que relacional | |
| `>=` | mayor o igual que relacional | |
| `instanceof` | comparación de tipos | |
| `==` | igual que relacional | izquierda a derecha |
| `!=` | no es igual que relacional | |
| `&` | AND a nivel de bits | izquierda a derecha |
| `^` | OR excluyente a nivel de bits | izquierda a derecha |
| | OR excluyente lógico booleano | |
| `\|` | OR incluyente a nivel de bits | izquierda a derecha |
| | OR incluyente lógico booleano | |
| `&&` | AND lógico | izquierda a derecha |
| `\|\|` | OR lógico | izquierda a derecha |
| `?:` | condicional ternario | derecha a izquierda |

**Figura C.3** Tabla de precedencia de operadores en Java. (Parte 1 de 2.)

<table>
<tr><th>Operador de Java</th><th>Tipo</th><th>Asociatividad</th></tr>
<tr><td>=</td><td>asignación</td><td>derecha a izquierda</td></tr>
<tr><td>+=</td><td>asignación de suma</td><td></td></tr>
<tr><td>-=</td><td>asignación de resta</td><td></td></tr>
<tr><td>*=</td><td>asignación de multiplicación</td><td></td></tr>
<tr><td>/=</td><td>asignación de división</td><td></td></tr>
<tr><td>%=</td><td>asignación de módulo</td><td></td></tr>
<tr><td>&=</td><td>asignación de AND a nivel de bits</td><td></td></tr>
<tr><td>^=</td><td>asignación de OR excluyente a nivel de bits</td><td></td></tr>
<tr><td>|=</td><td>asignación de OR incluyente a nivel de bits</td><td></td></tr>
<tr><td><<=</td><td>asignación de desplazamiento a la izquierda a nivel de bits</td><td></td></tr>
<tr><td>>>=</td><td>asignación de desplazamiento a la derecha a nivel de bits con extensión de signo</td><td></td></tr>
<tr><td>>>>=</td><td>asignación de desplazamiento a la derecha a nivel de bits con extensión de cero</td><td></td></tr>
</table>

**Figura C.3** Tabla de precedencia de operadores en Java. (Parte 2 de 2.)



# Conjunto de caracteres ASCII

**Conjunto de caracteres ASCII**

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| **1** | lf | vt | ff | cr | so | si | dle | dcl | dc2 | dc3 |
| **2** | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| **3** | rs | us | sp | ! | “ | # | $ | % | & | ‘ |
| **4** | ( | ) | * | + | , | - | . | / | 0 | 1 |
| **5** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| **6** | < | = | > | ? | @ | A | B | C | D | E |
| **7** | F | G | H | I | J | K | L | M | N | O |
| **8** | P | Q | R | S | T | U | V | W | X | Y |
| **9** | Z | [ | \ | ] | ^ | _ | ‘ | a | b | c |
| **10** | d | e | f | g | h | i | j | k | l | m |
| **11** | n | o | p | q | r | s | t | u | v | w |
| **12** | x | y | z | { | \| | } | ~ | del | | |

**Figura D.1** Conjunto de caracteres ASCII.

Los dígitos que se encuentran a la izquierda de la tabla son los dígitos a la izquierda del equivalente decimal (0-127) del código del carácter, y los dígitos que se encuentran en la parte superior de la tabla son los dígitos a la derecha del código del carácter. Por ejemplo, el código de carácter para “**F**” es 70, y el código del carácter para “**&**” es 38.






# Sistemas de numeración




## Objetivos

- Comprender los conceptos básicos de los sistemas de numeración tales como base, valor posicional y valor simbólico.
- Comprender cómo trabajar con números representados en los sistemas de numeración binario, octal y hexadecimal.
- Representar los números binarios como números octales o hexadecimales.
- Convertir números octales y hexadecimales en números binarios.
- Convertir números decimales en sus equivalentes binarios, octales y hexadecimales y viceversa.
- Comprender la aritmética binaria y cómo se representan los números binarios negativos mediante la notación de complemento a dos.

*Aquí sólo hay números ratificados.*
William Shakespeare

*La naturaleza tiene un cierto sistema aritmético-geométrico coordinado, ya que cuenta con todo tipo de modelos. Lo que experimentamos de la naturaleza es mediante modelos, y todos los modelos de la naturaleza son muy bellos.*

*Eso me indica que los sistemas de la naturaleza deben ser una verdadera belleza, ya que en la química encontramos que las asociaciones siempre se dan con hermosos números enteros; las fracciones no existen.*
Richard Buckminster Fuller

## Plan general



## E.1 Introducción

En este apéndice explicaremos la clave de los sistemas de numeración que utilizan los programadores en C, en especial cuando trabajan en proyectos de software que requieren una interacción cercana a "nivel del hardware". Los proyectos como estos incluyen sistemas operativos, software para redes de cómputo, compiladores, sistemas de bases de datos y aplicaciones que requieren un alto rendimiento.

Cuando escribimos un entero tal como 227 o –63 en un programa en C, asumimos que el número está en el *sistema de numeración decimal (base 10)*. Los *dígitos* en el sistema de numeración decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. El dígito de menor valor es el 0, y el dígito de mayor valor es el 9; uno menos que la base 10. En su interior, las computadoras utilizan el *sistema de numeración binario (base 2)*. El sistema de numeración binario sólo contiene 2 dígitos, a saber, 0 y 1. El dígito con menor valor es el 0, y el dígito con el valor más alto es el 1; uno menos que la base 2.

Como veremos más adelante, los números binarios tienden a ser más grandes que sus equivalentes en decimal. Los programadores que trabajan con lenguajes ensambladores y en lenguajes de alto nivel como C que permiten a los programadores alcanzar el "nivel de la máquina", encuentran que es conveniente trabajar con números binarios. De manera que los otros dos sistemas de numeración, los *sistemas de numeración octal (base 8)* y *hexadecimal (base 16)*, son muy populares, principalmente porque son convenientes para abreviar los números binarios.

En el sistema de numeración octal, el rango de los dígitos es de 0 a 7. Debido a que tanto el sistema de numeración binario como el octal contienen menos dígitos que el sistema de numeración decimal, sus dígitos corresponden a los dígitos del sistema decimal.

El sistema de numeración hexadecimal tiene un problema debido a que requiere dieciséis dígitos; un dígito con el valor más bajo, 0, y un dígito con el valor más alto equivalente al número decimal 15 (uno menos que la base 16). Por convención, utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales de 10 a 15. Así, en hexadecimal podemos tener números como 876 que consten solamente de dígitos parecidos a los decimales, números como 8A55F que consten de letras y dígitos, y números como FFE que consten solamente de letras. En algunas ocasiones, un número hexadecimal parece ser una palabra como BEBE o DEBE, esto puede parecer extraño para los programadores acostumbrados a trabajar con números. En las figuras E1 y E2 resumimos los dígitos de los sistemas de numeración binario, octal, decimal y hexadecimal.

Cada uno de estos sistemas de numeración utiliza una *notación posicional*. Cada posición en la que escribimos un dígito tiene un *valor posicional* diferente. Por ejemplo, en el número decimal 937 (al 9, al 3 y al 7 se les conoce como *valores de símbolo*), decimos que el 7 se escribe en la posición de las unidades, el tres se escribe en la posición de las decenas y el 9 se escribe en la posición de las centenas. Observe que cada una de estas posiciones es una potencia de la base (base 10) y que estas potencias comienzan con 0 y se incrementan en uno al desplazarnos hacia la izquierda del número (figura E3).

Para números decimales mayores, las siguientes posiciones a la izquierda serían: la posición de los miles (10 a la tercera potencia), la posición de los diez miles (10 a la cuarta potencia), la posición de los cien miles (10 a la quinta potencia), la posición de los millones (10 a la sexta potencia), la posición de los diez millones (10 a la séptima potencia), y así sucesivamente.

| Dígito binario | Dígito octal | Dígito decimal | Dígito hexadecimal |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| | 2 | 2 | 2 |
| | 3 | 3 | 3 |
| | 4 | 4 | 4 |
| | 5 | 5 | 5 |
| | 6 | 6 | 6 |
| | 7 | 7 | 7 |
| | | 8 | 8 |
| | | 9 | 9 |
| | | | A (valor decimal 10) |
| | | | B (valor decimal 11) |
| | | | C (valor decimal 12) |
| | | | D (valor decimal 13) |
| | | | E (valor decimal 14) |
| | | | F (valor decimal 15) |

**Figura E.1** Dígitos de los sistemas de numeración binario, octal, decimal y hexadecimal.

| Atributo | Binario | Octal | Decimal | Hexadecimal |
|---|---|---|---|---|
| Base | 2 | 8 | 10 | 16 |
| Dígito de menor valor | 0 | 0 | 0 | 0 |
| Dígito de mayor valor | 1 | 7 | 9 | F |

**Figura E.2** Comparación de los sistemas de numeración binario, octal, decimal y hexadecimal.

| Valores posicionales en el sistema de numeración decimal | | | |
|---|---|---|---|
| Dígito decimal | 9 | 3 | 7 |
| Nombre de posición | Centenas | Decenas | Unidades |
| Valor posicional | 100 | 10 | 1 |
| Valor posicional como una potencia de la base (10) | $10^2$ | $10^1$ | $10^0$ |

**Figura E.3** Valores posicionales en el sistema de numeración decimal.

En el número binario 101, decimos que el dígito 1 en la extrema derecha está escrito en la posición de los unos, el 0 está escrito en la posición de los dos y el 1 a la extrema izquierda está escrito en la posición de los cuatros. Observe que cada una estas posiciones es una potencia de la base (base 2), y que estas potencias comienzan en 0 y se incrementan en 1 mientras nos desplazamos hacia la izquierda del número (figura E.4).

Para números binarios más grandes, las siguientes posiciones a la izquierda serían: la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la posición de los treinta y dos (2 a la quinta potencia) la posición de los sesenta y cuatros (2 a la sexta potencia), y así sucesivamente.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos, el 2 se escribe en la posición de los ochos y el 4 se escribe en la posición de los sesenta y cuatros. Observe que cada una de estas posiciones es una potencia de la base (base 8), y que estas potencias comienzan en 0 y se incrementan en 1 mientras nos desplazamos a la izquierda del número (figura E.5).

Para números octales más grandes, las siguientes posiciones a la izquierda serían: la posición de los quinientos doces (8 a la tercera potencia), la posición de los cuatro mil noventa y seis (8 a la cuarta potencia), la posición de los treinta y dos mil setecientos sesenta y ochos (8 a la quinta potencia), y así sucesivamente.

En el número hexadecimal 3DA, decimos que A se escribe en la posición de los unos, la D se escribe en la posición de los dieciséis y 3 se escribe en la posición de los doscientos cincuenta y seises. Observe que cada una de estas posiciones es una potencia de la base (base 16), y que estas potencias comienzan en 0 y se incrementan en 1 mientras nos desplazamos a la izquierda del número (figura E.6).

Para números hexadecimales más grandes, las siguientes posiciones a la izquierda serían: la posición de los cuatro mil noventa y seises (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seises (16 a la 4a potencia), y así sucesivamente.

**Valores posicionales en el sistema de numeración binario**

| | | | |
|---|---|---|---|
| Dígito binario | **1** | **0** | **1** |
| Nombre de la posición | Cuatros | Dos | Unos |
| Valor posicional | **4** | **2** | **1** |
| Valor posicional como una potencia de la base (2) | $2^2$ | $2^1$ | $2^0$ |

**Figura E.4** Valores posicionales en el sistema de numeración binario.

**Valores posicionales en el sistema de numeración octal**

| | | | |
|---|---|---|---|
| Dígito decimal | **4** | **2** | **5** |
| Nombre de la posición | Sesenta y cuatros | Ochos | Unos |
| Valor posicional | **64** | **8** | **1** |
| Valor posicional como una potencia de la base (8) | $8^2$ | $8^1$ | $8^0$ |

**Figura E.5** Valores posicionales en el sistema de numeración octal.

**Valores posicionales en el sistema de numeración hexadecimal**

| | | | |
|---|---|---|---|
| Dígito decimal | **3** | **D** | **A** |
| Nombre de la posición | Doscientos cincuenta y seis | Dieciséis | Unos |
| Valor posicional | **256** | **16** | **1** |
| Valor posicional como una potencia de la base (16) | $16^2$ | $16^1$ | $16^0$ |

**Figura E.6** Valores posicionales en el sistema de numeración hexadecimal.

## E.2 Cómo expresar números binarios en números octales y números hexadecimales

El principal uso de los números octales y hexadecimales en computación es para abreviar las largas representaciones de los números binarios. En la figura E.7 resaltamos el hecho de que en los sistemas de numeración, los números binarios muy grandes pueden expresarse de manera concisa con sistemas de numeración con bases más altas que el sistema de numeración binario.

Una relación particularmente importante que tanto el sistema de numeración octal como el sistema de numeración hexadecimal tienen con el sistema binario es que las bases en octal y hexadecimal (8 y 16 respectivamente) son potencias de la base del sistema de numeración binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes en octal y hexadecimal. Vea si puede determinar la manera en que esta relación facilita la expresión de números binarios en octal y en hexadecimal. La respuesta radica en los números.

**Binario NúmeroOctal equivalenteDecimal equivalente**
**100011010001432l8D1**

Para ver cómo el número binario se convierte fácilmente en octal, simplemente divida el número binario de 12 dígitos en grupos de tres bits consecutivos cada uno, y escriba dichos grupos sobre los dígitos correspondientes a los números octales, de la siguiente manera:

```
100 011 010 001
4   3   2   1
```

Observe que el dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de dicho número binario de tres dígitos como lo mostramos en la figura E.7.

Se puede observar el mismo tipo de relación al hacer la conversión de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos cada uno, y escriba dichos grupos sobre los dígitos correspondientes al número hexadecimal, de la siguiente manera

```
100011010001
8   D   1
```

| Número decimal | Representación binaria | Representación octal | Representación hexadecimal |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |

**Figura E.7** Equivalentes binarios, octales y hexadecimales del sistema de numeración decimal.

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de cuatro dígitos, como lo mostramos en la figura E.7.

## E.3 Conversión de números octales y números hexadecimales a números binarios

En la sección anterior, vimos cómo convertir números binarios en sus equivalentes octales y hexadecimales, formando grupos de dígitos binarios y rescribiendo estos grupos como sus equivalentes en dígitos octales o dígitos hexadecimales. Este proceso puede utilizarse de manera inversa para producir el equivalente binario de un número octal o hexadecimal.

Por ejemplo, el número octal 653 se convierte a binario simplemente escribiendo el 6 como el valor equivalente binario de tres dígitos 110, el 5 como el equivalente binario de tres dígitos 101 y el 3 como el equivalente binario de tres dígitos 011, para formar el número binario de nueve dígitos 110101011.

El número hexadecimal FAD5 se convierte a binario simplemente escribiendo la F como su equivalente binario de cuatro dígitos 1111, la A como su equivalente binario de cuatro dígitos 1010, la D como su equivalente binario de cuatro dígitos 1101 y el 5 como su equivalente binario de cuatro dígitos 0101, para formar el número de 16 dígitos 1111101011010101.

## E.4 Conversión de números binarios, octales o hexadecimales a números decimales

Debido a que estamos acostumbrados a trabajar en decimal, con frecuencia es conveniente convertir un número binario, octal o hexadecimal a decimal para tener la idea de lo que la computadora hace en “realidad”. Nuestros diagramas de la sección E.1 expresan los valores posicionales en decimal. Para convertir un número a decimal desde otra base, multiplique el equivalente decimal de cada dígito por su valor posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el decimal 53 como muestra la figura E.8.

Para convertir el octal 7614 al decimal 3980, utilizamos la misma técnica, esta vez mediante los valores posicionales octales que muestra la figura E.9.

Para convertir el hexadecimal AD3B al decimal 44347, utilizamos la misma técnica, esta vez mediante los valores posicionales hexadecimales adecuados que muestra la figura E.10.

**Conversión de un número binario a decimal**

| | | | | | | |
|---|---|---|---|---|---|---|
| Valores posicionales: | 32 | 16 | 8 | 4 | 2 | 1 |
| Valores de símbolos: | 1 | 1 | 0 | 1 | 0 | 1 |
| Productos: | 1*32=32 | 1*16=16 | 0*8=0 | 1*4=4 | 0*2=0 | 1*1=1 |
| Suma: | = 32 + 16 + 0 + 4 + 0 + 1 = 53 | | | | | |

**Figura E.8** Conversión de un número binario a decimal.

**Conversión de un número octal a decimal**

| | | | | |
|---|---|---|---|---|
| Valores posicionales: | 512 | 64 | 8 | 1 |
| Valores de símbolos: | 7 | 6 | 1 | 4 |
| Productos: | 7*512=3584 | 6*64=384 | 1*8=8 | 4*1=4 |
| Suma: | = 3584 + 384 + 8 + 4 = 3980 | | | |

**Figura E.9** Conversión de un número octal a decimal.

**Conversión de un número hexadecimal a decimal**

| | | | | |
|---|---|---|---|---|
| Valores posicionales: | **4096** | **256** | **16** | **1** |
| Valores de símbolos: | **A** | **D** | **3** | **B** |
| Productos: | **A*4096=40960** | **D*256=3328** | **3*16=48** | **B*1=11** |
| Suma: | **= 40960 + 3328 + 48 + 11 = 44347** | | | |

**Figura E.10** Conversión de un número hexadecimal a decimal.

## E.5 Conversión de números decimales a números binarios, octales o hexadecimales

Las conversiones de la sección E.4 siguen de manera natural las convenciones de la notación posicional. La conversión posicional de decimal a binario, octal, o hexadecimal también siguen estas convenciones.

Suponga que deseamos convertir el decimal 57 a binario. Comenzamos por escribir los valores posicionales de las columnas de derecha a izquierda hasta que alcanzamos la columna cuyo valor posicional es mayor que el número decimal. No necesitamos dicha columna, de modo que la descartamos. Entonces, primero escribimos:

Valores posicionales:**6432168421**

Luego descartamos la columna con el valor posicional 64 y dejamos:

Valores posicionales:**3216842 1**

A continuación, trabajamos desde la columna en la extrema izquierda hacia la derecha. Dividimos 57 entre 32 y observamos que existe un 32 en 57 con un residuo de 25, de modo que escribimos 1 en la columna de 32. Dividimos 25 entre 16 y observamos que existe un 16 en 25 con un residuo de 9 y escribimos 1 en la columna de 16. Dividimos 9 entre 8 y observamos que existe un 8 en 9 con un residuo de 1. Las siguientes dos columnas producen cocientes de cero cuando los valores posicionales se dividen entre 1, de modo que escribimos 0s en las columnas de 4 y de 2. Por último, 1 entre 1 es 1, de modo que escribimos 1 en la columna de 1. Esto arroja:

Valores posicionales:**3216842 1**
Valores de símbolos:**11 10 0 1**

y así, el decimal 57 es equivalente al binario 111001.

Para convertir el decimal 103 a octal, comenzamos por escribir los valores posicionales de las columnas hasta que alcanzamos una columna cuyo valor posicional sea mayor que el número decimal. No necesitamos dicha columna, de modo que la descartamos. Entonces, primero escribimos:

Valores posicionales:**5126481**

Luego descartamos la columna con el valor posicional 512, y tenemos:

Valores posicionales:**6481**

A continuación, trabajamos desde la columna en la extrema izquierda hacia la derecha. Dividimos 103 entre 64 y observamos que existe un 64 en 103 con un residuo de 39, de modo que escribimos 1 en la columna de 64. Dividimos 39 entre 8 y observamos que existen cuatro 8s en 39 con un residuo de 7 y escribimos 4 en la columna de 8. Por último, dividimos 7 entre 1 y observamos que existen 7 unos en 7 sin residuo, de modo que escribimos 7 en la columna de 1. Esto arroja:

Valores posicionales:**6481**
Valores de símbolos:**14 7**

y así, el decimal 103 es equivalente al octal 147.

Para convertir el decimal 375 a hexadecimal, comenzamos por escribir los valores posicionales de las columnas hasta que alcanzamos una columna cuyo valor posicional sea mayor que el número decimal. No necesitamos dicha columna, de manera que la descartamos. Entonces, primero escribimos

Valores posicionales:**4096256161**

Después descartamos la columna con el valor posicional 4096, y tenemos:

Valores posicionales:**256161**

A continuación, trabajamos desde la columna en la extrema izquierda hacia la derecha. Dividimos 375 entre 256 y observamos que existe un 256 en 375 con un residuo de 119, por lo que escribimos 1 en la columna de 256. Dividimos 119 entre 16 y observamos que existen siete 16s en 119 con un residuo de 7, y escribimos 7 en la columna de 16. Por último, dividimos 7 entre 1 y observamos que existen siete unos en 7 sin residuo, de modo que escribimos 7 en la columna de 1. Esto arroja:

Valores posicionales:**256161**
Valores de símbolos:**17 7**

y así, el decimal 375 es equivalente al hexadecimal 177.

## E.6 Números binarios negativos: Notación de complemento a dos

La explicación de este apéndice se ha enfocado en números positivos. En esta sección, explicamos cómo las computadoras representan números negativos mediante el uso de la *notación de complemento a dos*. Primero, explicamos cómo se forma el complemento a dos de un número binario y luego mostramos por qué representa el *valor negativo* del número binario dado.

Considere una máquina con enteros de 32 bits. Suponga

```
int valor = 13;
```

La representación a 32 bits de **valor** es

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de **valor** primero formamos su *complemento a uno*, aplicando el *operador de complemento a nivel de bits (~)*:

```
complementoAUnoDelValor = ~valor;
```

Internamente, **~valor** ahora es **valor** con cada uno de sus bits invertidos, los unos se convierten en ceros y los ceros se convierten en unos, de la siguiente manera:

```
valor:
00000000 00000000 00000000 00001101

~valor (es decir, valores de complemento a uno):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de **valor** simplemente sumamos uno al complemento a uno de **valor**. Entonces,

```
Complemento a dos de valor:
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a –13, debemos ser capaces de sumarlo al número 13 y obtener un resultado de 0. Intentemos esto:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
------------------------------------
 00000000 00000000 00000000 00000000
```

Descartamos el bit de acarreo de la columna de la extrema izquierda, y obtenemos cero como resultado. Si a un número le sumamos su complemento a uno, el resultado sería todo en 1s. La clave para obtener el resultado en ceros es que el complemento a dos es 1 mayor que el complemento a uno. La suma de 1 provoca que cada columna sume 0 con un acarreo de 1. El acarreo se mantiene en movimiento hacia la izquierda desde el bit de la extrema izquierda, y por ello el número resultante es cero.

En realidad, las computadoras realizan una resta como

```
x = a – valor;
```

al sumar el complemento a dos de **valor** a **a**, de la siguiente manera:

```
x = a + (~valor + 1);
```

Suponga que, como antes, **a** es 27 y **valor** es 13. Si el complemento a dos de **valor** es en realidad el valor negativo de **valor**, entonces sumarle a **a** el complemento a dos de **valor** debe producir el resultado 14. Intentemos esto:

```
a (es decir, 27)  00000000 00000000 00000000 00011011
 +(~valor + 1)+ 11111111 11111111 11111111 11110011
              -----------------------------
              00000000 00000000 00000000 00001110
```

lo cual es igual a 14.

## RESUMEN

- Asumimos que un entero en un programa en C como 19, 227, o −63, se encuentra en el sistema de numeración decimal (base 10). Los dígitos del sistema de numeración decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. El dígito con menos valor es el 0, y el dígito con el mayor valor es el 9, uno menos que la base 10.
- De manera interna, las computadoras utilizan el sistema de numeración binario (base 2); dicho sistema de numeración solamente contiene dos dígitos, a saber 0 y 1. Su dígito de menor valor es el 0 y su dígito de mayor valor es el 1, uno menos que la base 2.
- El sistema de numeración octal (base 8) y el sistema de numeración hexadecimal (base 16) son populares primordialmente porque son convenientes para abreviar los números binarios.
- Los dígitos del sistema de numeración octal se encuentran en el rango de 0 a 7.
- El sistema de numeración hexadecimal tiene un problema, ya que requiere de 16 dígitos; el dígito de menor valor es el 0 y el dígito de mayor valor es el equivalente al número 15 en decimal (uno menos que la base 16). Por convención, utilizamos las letras de la A a la F para representar los dígitos decimales que corresponden a los valores entre 10 y 15.
- Cada sistema de numeración utiliza una notación posicional; cada posición en la que se escribe un dígito tiene un valor posicional diferente.
- Una relación particularmente importante que tanto el sistema de numeración octal como el sistema de numeración decimal tienen con respecto al sistema binario es que las bases octal y hexadecimal (8 y 16 respectivamente) son potencias de la base del sistema de numeración binario (base 2).
- Para convertir un número octal a un número binario, reemplace cada dígito octal con su equivalente binario de tres dígitos.
- Para convertir un número binario a un número hexadecimal, reemplace cada dígito hexadecimal con su equivalente binario de cuatro dígitos.
- Debido a que estamos acostumbrados a trabajar en decimal, es conveniente convertir un número binario, octal o hexadecimal a decimal, para entender el sentido “real” de un número.
- Para convertir un número a decimal desde otra base, multiplique el equivalente decimal de cada dígito por su valor posicional, y sume el valor de estos productos.
- Las computadoras representan los valores negativos mediante la notación de complemento a dos.
- Para formar el negativo de un valor en binario, primero forme el complemento a uno mediante la aplicación del operador de complemento a nivel de bits de C (**~**). Esto invierte los bits del valor. Para formar el complemento a dos de un valor, simplemente sume uno al complemento a uno del valor.

## TERMINOLOGÍA

base
conversiones
dígito
notación de complemento a dos
notación de complemento a uno
notación posicional
operador de complemento a nivel de bits (~)
sistema de numeración binario
sistema de numeración de base 10
sistema de numeración de base 16
sistema de numeración de base 2
sistema de numeración de base 8
sistema de numeración decimal
sistema de numeración hexadecimal
sistema de numeración octal
valor del símbolo
valor negativo
valor posicional

## EJERCICIOS DE AUTOEVALUACIÓN

**E.1** Las bases para los sistemas de numeración decimal, binario, octal y hexadecimal son ______, _______, y ________ respectivamente.

**E.2** Por lo general, las representaciones decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos que los que contiene un número binario.

**E.3** (*Verdadero*/*Falso.*) Una razón muy popular para utilizar el sistema de numeración decimal es que conforma una notación conveniente para abreviar números binarios con la simple sustitución de un dígito decimal por cada grupo de cuatro bits binarios.

**E.4** La representación (octal/decimal/hexadecimal) de un valor binario grande es la más concisa (de las alternativas dadas).

**E.5** (*Verdadero*/*Falso.*) El dígito de más alto valor en cualquier base es uno más que la base.

**E.6** (*Verdadero*/*Falso.*) El dígito con el valor más bajo es uno menos que la base.

**E.7** El valor posicional del dígito a la extrema derecha de cualquier número, ya sea en binario, octal, decimal o hexadecimal, siempre es _________.

**E.8** El valor posicional del dígito a la izquierda del dígito a la extrema derecha de cualquier número, ya sea en binario, octal, decimal o hexadecimal, siempre es igual a _________.

**E.9** Complete los valores que faltan en la siguiente tabla de valores posicionales para las cuatro posiciones a la extrema derecha de cada uno de los sistemas de numeración indicados:

| | | | | |
|---|---|---|---|---|
| decimal | **1000** | **100** | **10** | **1** |
| hexadecimal | **...** | **256** | **...** | **...** |
| binario | **...** | **...** | **...** | **...** |
| octal | **512** | **...** | **8** | **...** |

**E.10** Convierta el binario 110101011000 a octal y a hexadecimal.

**E.11** Convierta el hexadecimal BEBA a binario.

**E.12** Convierta el octal 7316 a binario.

**E.13** Convierta el hexadecimal 4FEC a octal. [*Pista:* Primero convierta 4FEC a binario, y luego convierta el binario a octal.]

**E.14** Convierta el binario 1101110 a decimal.

**E.15** Convierta el octal 317 a decimal.

**E.16** Convierta el hexadecimal EFD4 a decimal.

**E.17** Convierta el decimal 177 a binario, a octal y a hexadecimal.

**E.18** Muestre la representación en binario del decimal 417. Luego muestre el complemento a uno de 417 y el complemento a dos de 417.

**E.19** ¿Cuál es el resultado cuando se suma el complemento a dos de un número a sí mismo?

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

**E.1** 10, 2, 8,16.

**E.2** Menos.

**E.3** Falso.

**E.4** Hexadecimal.

**E.5** Falso. El dígito de mayor valor en cualquier base es uno menos que la base.

**E.6** Falso. El dígito de menor valor en cualquier base es cero.

**E.7** 1 (la base elevada a la potencia 0).

**E.8** La base del sistema de numeración.

**E.9**

| | | | | |
|---|---|---|---|---|
| decimal | **1000** | **100** | **10** | **1** |
| hexadecimal | **4096** | **256** | **16** | **1** |
| binario | **8** | **4** | **2** | **1** |
| octal | **512** | **64** | **8** | **1** |

**E.10** Octal 6530; hexadecimal D58.

**E.11** Binario 1011 1110 1011 1010.

**E.12** Binario 111 011 001 110.

**E.13** Binario 0 100 111 111 101 100; octal 47754.

**E.14** Decimal 2+4+8+32+64=110.

**E.15** Decimal 7+1*8+3*64=7+8+192=207.

**E.16** Decimal 4+13*16+15*256+14*4096=61396.

**E.17** Decimal 177
a binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

a octal

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

a hexadecimal

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

**E.18** Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+
(1*1)
110100001
```

```
Complemento a 1: 001011110
Complemento a 2: 001011111
Verificación: número binario original + su complemento a dos
```

```
110100001
001011111
---------
000000000
```

**E.19** Cero.

## *EJERCICIOS*

**E.20** Algunas personas argumentan que muchos de nuestros cálculos serían más fáciles en el sistema de numeración con base 12, debido a que 12 es divisible por muchos más números que 10 (por la base 10). ¿Cuál es el dígito de menor valor en la base 12? ¿Cuál podría ser el símbolo del valor más alto en la base 12? ¿Cuáles son los valores posicionales de las cuatro posiciones a la extrema derecha de cualquier número en el sistema de numeración con base 12?

**E.21** ¿Cómo se relaciona el mayor valor de símbolo de los sistemas de numeración que hemos explicado con el valor posicional del primer dígito a la izquierda del dígito a la extrema derecha de cualquier número en estos sistemas de numeración?

**E.22** Complete la siguiente tabla de valores posicionales para las cuatro posiciones a la extrema derecha de cada uno de los sistemas de numeración indicados.

| | | | | |
|---|---|---|---|---|
| decimal | **1000** | **100** | **10** | **1** |
| base 6 | **...** | **...** | **6** | **...** |
| base 13 | **...** | **169** | **...** | **...** |
| base 3 | **27** | **...** | **...** | **...** |

**E.23** Convierta el binario 1001011111010 a octal y a hexadecimal.

**E.24** Convierta el hexadecimal 3A7D a binario.

**E.25** Convierta el hexadecimal 765F a octal. [*Pista:* Primero convierta 765F a binario, y luego convierta ese número binario a octal.]

**E.26** Convierta el binario 1011110 a decimal.

**E.27** Convierta el octal 426 a decimal.

**E.28** Convierta el hexadecimal FFFF a decimal.

**E.29** Convierta el decimal 299 a binario, a octal y a hexadecimal.

**E.30** Muestre la representación binaria del decimal 779. Luego muestre el complemento a uno de 779 y el complemento a dos de 779.

**E.31** ¿Cuál es el resultado cuando a un número se le suma su complemento a dos?

**E.32** Muestre el complemento a dos del valor entero $-1$ en una máquina con enteros de 32 bits.



# Recursos de la biblioteca estándar de C

Este apéndice contiene una lista de valiosos recursos para la biblioteca estándar de C en Internet y en la World Wide Web. Estas funciones, tipos y macros son definidos por el American National Standards Institute y están diseñados para garantizar la portabilidad entre los sistemas operativos y para incrementar su eficiencia. Aunque no son parte del lenguaje C, cualquier compilador que soporte el C de ANSI, por lo general proporcionará las definiciones para estas bibliotecas.

En 1999, la Intenational Standards Organization aprobó una nueva versión de C, conocida como C99. Esta versión soporta cualquiera de las características descritas en el apéndice B. Muchos de los recursos listados abajo proporcionan información acerca de las adiciones de C99 a la biblioteca estándar de C.

Para mayor información acerca de ANSI o para adquirir los documentos del estándar, visite ANSI en **`www.ansi.org`**.

## F.1 Recursos para la biblioteca estándar de C

**`www.ansi.org`**
Aquí puede encontrar y adquirir todos los documentos de ANSI, incluso el estándar de C99.

**`www.incits.org`**
El InterNational Committe of Information Technology Support es el grupo de asesoramiento tecnológico de ANSI para el ISO/IEC Joint Technical Committee 1. Aquí puede encontrar y adquirir el estándar de C99.

**`msdn.microsoft.com/visualc/`**
La página de Visual C++ contiene vínculos hacia muchos grupos de noticias, foros de discusión y sitios relacionados, así como información acerca del soporte para C/C++ y mejoras.

**`www.dinkumware.com/libraries_ref.html`**
Las licencias de dinkumware de las bibliotecas de C y C++ cumplen con los estándares de ANSI y proporcionan documentación en línea.

**`ccs.ucsd.edu/c/`**
Un sitio integral para el C estándar provisto por la Universidad de California en San Diego.

**`www.acm.uiuc.edu/webmonkeys/book/c_guide/`**
Una referencia sobre la biblioteca de C colocada por la comunidad de la Association of Computing Machinery de la Universidad de Illinois.

**`www.thefreecountry.com/compilers/cpp.shtml`**
Este sitio Web ofrece bibliotecas de C/C++ gratuitas, editores, IDEs, compiladores y libros.

**`www.freeprogrammingresources.com/cpplib.html`**
Este sitio Web ofrece muchos recursos gratuitos de programación que incluyen bibliotecas de C y C++.

**www.programmersheaven.com/**
Un completo conjunto de recursos en cualquier lenguaje y ambiente para programadores. Este sitio también ofrece herramientas y bibliotecas para C y C++.

**www.lysator.liu.se/c/**
Una colección de artículos y libros relacionados con la historia de C y el estándar de ANSI.

P. J. Plauger, representante del comité responsable del desarrollo del C estándar de ISO, ha escrito varios libros acerca de las bibliotecas estándares así como de otros temas de programación. También estuvo involucrado en el desarrollo del estándar de C++ y ha escrito libros acerca de sus bibliotecas. Algunos de sus trabajos incluyen:

- *The Standard C Library*, P. J. Plauger, Prentice Hall, 1993-1994.
- *Standard C: A Reference*, P. J. Plauger y Jim Brodie, Prentice Hall, 1996.
- *The Drafts Standard C++ Library*, P. J. Plauger, Prentice Hall, 1995.

# Índice

# B

# F

## G



## H

# I



# J

## K



## L



## LL



## M

## N

## P

## Q



## R

## S



## W

## Acuerdo para el usuario

Al abrir este paquete usted acepta lo siguiente:
No puede copiar ni redistribuir el contenido del CD-ROM en su totalidad. La copia y redistribución de cada uno de los programas está sujeta a los términos establecidos por los respectivos poseedores de los derechos de autor, esto incluye el programa de instalación y el código anexo.

El software se proporciona como está, sin ninguna garantía de ninguna clase, ni implícita ni explícita, incluyendo (pero sin limitarse) a las garantías implícitas de comercialización y adecuación a un propósito particular. Ni el editor ni sus vendedores o distribuidores asumen responsabilidad alguna por ningún daño, supuesto o real, directo o indirecto, que surja por el uso de este software. Esto incluye, sin limitarse, la interrupción del servicio, pérdida de datos, pérdida de tiempo en clase, pérdida de beneficios por consultoría o cualquier otro derivado del uso de estos programas. El uso de este software está sujeto a los términos de la Licencia de Código Binario y a las condiciones del Contrato de licencia para el usuario final del software de Microsoft que se incluye en las páginas finales de este libro, el cual le pedimos lea cuidadosamente.

## Cómo explorar el CD-ROM

Si tiene activada la característica de reproducción automática (AutoPlay), su computadora ejecutará automáticamente la interfaz del CD-ROM; de otra manera haga doble clic en el archivo **`AUTORUN.EXE`.**

## Contenido del CD-ROM

- Microsoft® Visual C++® 6.0 Introductory Edition.
- Todo el código utilizado en los ejercicios del libro.
- Vínculos hacia los sitios Web mencionados en el libro.

## Requerimientos de sistema para ejecutar Microsoft® Visual C++® 6.0 Introductory Edition

- PC con procesador tipo Pentium, a 90 MHz o superior.
- Microsoft Windows 95 o posterior o Microsoft Windows NT 4.0 con Service Pack 3 o posterior (que incluya el Service Pack 3).
- Microsoft Internet Explorer 4.01 (con Service Pack 1 incluido) o posterior.
- Unidad de CD-ROM.
- Monitor VGA (o superior) con alta resolución; Super VGA recomendado.
- Mouse de Microsoft o dispositivo apuntador compatible.
- 25 MB de RAM para Windows 95 o posterior (48 MB recomendado).
- Espacio en disco requerido para la instalación básica de VC++: Típica, 266 MB; Máxima, 370 MB.
- Espacio en disco requerido para instalar VC++ y Service Pack: 345 MB (Nota: Dado que el Service Pack reemplaza la mayoría de los archivos, sólo se necesitan un poco más de 30 MB).
- Conexión a Internet.

## Usuarios de Windows 2000 y XP

Microsoft Visual C++ 6.0 Introductory Edition puede ser instalado en cualquiera de estas versiones de Windows; pero necesita acceder al sistema con privilegios de "administrador".
*Nota:* No se ofrece soporte para Windows 95.